

Runtime Verification

Grigore Roşu

University of Illinois at Urbana-Champaign

Contents

1	Introduction	5
1.1	A Taxonomy of Runtime Analysis Techniques	16
1.1.1	Trace Storing versus Non-Storing Algorithms	16
1.1.2	Synchronous versus Asynchronous Monitoring	18
1.1.3	Predictive versus Exact Analysis	19
2	Background, Preliminaries, Notations	21
2.1	Preliminaries	24
2.1.1	Membership Equational Logic	24
2.1.2	Maude	28
3	Safety Properties	35
3.1	Finite Traces	38
3.2	Infinite Traces	45
3.3	Finite and Infinite Traces	48
3.4	“Always Past” Characterization	51
4	Monitoring	55
4.1	Specifying Safety Properties as Monitors	55
4.2	Complexity of Monitoring a Safety Property	60
4.3	Monitoring Safety Properties is Arbitrarily Hard	66
4.4	Canonical Monitors	68
5	Event/Trace Observation	71
6	Monitor Synthesis: Finite State Machines (FSM)	73
6.1	Binary Transition Trees (BTT)	73
6.1.1	Multi-Transition and Binary Transition Tree Finite State Machines	73

6.1.2	From MT-FSMs to BTT-FSMs	78
6.2	Multi-Transitions and Binary Transition Trees	81
6.3	Binary Transition Tree Finite State Machines	84
7	Monitor Synthesis: Extended Regular Expressions (ERE)	87
7.1	Monitoring ERE Safety Needs Non-Elementary Space	87
7.1.1	Discussion and Relevance of the Lower-Bound Result	88
7.1.2	The Lower-Bound Result	94
7.2	Generating Optimal Monitors for ERE	100
7.2.1	Introduction	101
7.2.2	Extended Regular Expressions and Derivatives	104
7.2.3	Hidden Logic and Coinduction	106
7.2.4	Behavioral Equivalence, Satisfaction and Specification	107
7.2.5	Generating Minimal DFA Monitors by Coinduction	113
7.2.6	Implementation and Evaluation	115
8	Monitor Synthesis: Linear Temporal Logic (LTL)	121
8.1	Finite Trace Future Time Linear Temporal Logic	121
8.1.1	Finite Trace Semantics	122
8.1.2	Finite Trace Semantics in Maude	123
8.2	A Backwards, Asynchronous, but Efficient Algorithm	125
8.2.1	An Example	125
8.2.2	Generating Dynamic Programming Algorithms	128
8.3	A Forwards and Often Synchronous Algorithm	131
8.3.1	An Event Consuming Algorithm	131
8.3.2	Correctness and Completeness	136
8.3.3	Further Optimization by Memoization	138
8.4	Generating Forwards, Synchronous and Efficient Monitors	140
8.4.1	From LTL Formulae to BTT-FSMs	141
8.4.2	Examples	146
8.5	Conclusions	148
9	Efficient Monitoring of ω-Languages	151
9.1	Introduction	151
9.2	Preliminaries: Büchi Automata	155
9.3	Generating a <i>BTT-FSM</i> from a Büchi automaton	156
9.4	Monitor Generation and MOP	160
9.4.1	Evaluation	161
9.5	Conclusions	162

10 Efficient Monitoring of “Always Past” Temporal Safety	169
10.1 Introduction	170
10.2 The PathExplorer Architecture	173
10.2.1 The Observer	173
10.2.2 Code Instrumentation	174
10.3 Finite Trace Past Time LTL	175
10.3.1 Syntax	176
10.3.2 Formal Semantics	177
10.3.3 Recursive Semantics	178
10.3.4 Equivalent Logics	179
10.4 Monitoring Safety by Rewriting	181
10.4.1 Maude	182
10.4.2 Formulae and Data Structures	183
10.4.3 Propositional Calculus	184
10.4.4 Past Time Linear Temporal Logic	185
10.4.5 Monitoring with Maude	189
10.5 Synthesizing Monitors for Safety Properties	190
10.5.1 The Algorithm Illustrated by an Example	191
10.5.2 The Algorithm Formalized	194
10.5.3 Optimizing the Generated Code	198
10.5.4 Implementation of Offline and Inline Monitoring	200
10.6 Conclusion	205
10.7 Optimal Monitoring of “Always Past” Temporal Safety	206
10.7.1 The Monitor Synthesis Algorithm	206
10.7.2 A Maude Implementation of the Monitor Synthesizer	209
11 Monitoring “Always-Past” Temporal Safety with Call-Return	217
11.1 Introduction	218
11.2 PTLTL and PTCARET	220
11.3 PTCARET Derived Operators and Examples	226
11.4 A Monitor Synthesis Algorithm for PTCARET	228
11.4.1 The Target Language	229
11.4.2 The Monitor Synthesis Algorithm	231
11.4.3 Implementation as Logic Plugin, Optimizations, Example	234
11.5 Conclusion and Future Work	238
11.6 Auxiliary Material - not included in original paper	239
11.6.1 The Algorithm Formalized	242

12 Efficient Monitoring of Context-Free Patterns	249
13 Efficient Monitoring with Deterministic String Rewrite Systems	251
13.1 Introduction	252
13.1.1 Examples	253
13.1.2 Contributions	256
13.1.3 Paper Outline	257
13.2 Related Work and MOP	257
13.3 Monitoring SRS Specifications	259
13.3.1 Preliminaries	260
13.3.2 String Rewriting Algorithm Overview	260
13.3.3 Pattern Match Automata	261
13.3.4 Rewriting using Pattern Match Automata	267
13.4 Evaluation	273
13.5 Conclusion	276
14 Parametric Property Monitoring	277
15 Predictive Runtime Analysis	279
16 Static Analysis to Improve Runtime Verification	281
17 Semantics-Based Runtime Verification	283
17.1 Defining a Formal Semantics	283
17.2 Semantics-Based Symbolic Execution	283
17.3 Program Verification as Exhaustive Runtime Verification . . .	283
18 Conclusion and Future Work	285
18.1 Safety Properties and Monitoring	285

Topics to cover:

- Safety properties and their monitoring. How many safety properties are there? Can they all be monitored? Complexity of monitoring in different formalism: LTL, RE, ERE, CFG, SRS. Both dynamic properties, like above, and static properties (e.g., complex heap patterns).
- Event/Trace observation. How to observe the execution of a program? Instrumentation vs. runtime environment.
- Monitor synthesis. Generating optimal monitors for several formalisms: LTL, FT/PT-LTL, RE, ERE, CFG, SRS, PT-CaRet, Allen TL.
- Parametric property monitoring. How to deal with multiple instances of monitors?
- Predictive runtime analysis. Vector clock vs SMT-based techniques.
- Static analysis to improve runtime verification. Improve runtime overhead by not instrumenting what is unnecessary. Improve prediction capability by looking beyond the trace.
- Semantics-based Runtime Verification. Defining a formal language semantics. Using a semantics to do symbolic execution and runtime verify properties. Ultimate goal: verify programs by exhaustive runtime verification.

Papers which have been completely absorbed:

- * [164]: Section 7.2; Chapter 2
- * [154]: Chapter 2; Chapter 3; Chapter 4; Section 7.1;
- * [150]: Section 6.1; Chapter 8;
- * [54]: Chapter 6; Chapter 9;
- * [97, 96]: Chapter 10;
- * [155]: Chapter 11

Chapter 1

Introduction

Begin of intro stuff for chapter on safety

From SACS: *Abstract sacs:* *This paper addresses the problem of runtime verification from a foundational perspective, answering questions like “Is there a consensus among the various definitions of a safety property?” (Answer: Yes), “How many safety properties exist?” (Answer: As many as real numbers), “How difficult is the problem of monitoring a safety property?” (Answer: Arbitrarily complex), “Is there any formalism that can express all safety properties?” (Answer: No), etc. Various definitions of safety properties as sets of execution traces have been proposed in the literature, some over finite traces, others over infinite traces, yet others over both finite and infinite traces. By employing cardinality arguments and a novel notion of persistence, this paper first establishes the existence of bijective correspondences between the various notions of safety property. It then shows that safety properties can be characterized as “always past” properties. Finally, it proposes a general notion of monitor, which allows to show that safety properties correspond precisely to the monitorable properties, and then to establish that monitoring a safety property is arbitrarily hard.*

From safety: *Abstract safety:* Various definitions of safety properties as sets of execution traces have been introduced in the literature, some over finite traces, others over infinite traces, yet others over both finite and infinite traces. By employing cardinality arguments, this paper first shows that these notions of safety are ultimately equivalent, by showing each of them to have the cardinal of the continuum. It is then shown that all safety properties can be characterized as “always past” properties, and then that the problem of monitoring a safety property can be arbitrarily hard. Finally, two decidable specification formalisms for safety properties are discussed, namely extended regular expressions and past time LTL. It is shown that monitoring the former requires non-elementary space. An optimal monitor synthesis algorithm is given for the latter; the generated monitors run in space linear with the number of temporal operators and in time linear with the size of the formula.

A *safety property* is a behavioral property which, once violated, cannot be satisfied anymore. For example, a property “always $x > 0$ ” is violated when $x \leq 0$ is observed for the first time; this safety property remains violated even though eventually $x > 0$ might hold. That means that one can identify each safety property with a set of “bad” finite execution traces, with the intuition that once one of those is reached the safety property is violated.

There are several apparently different ways to formalize safety. Perhaps the most immediate one is to complement the “bad traces” above and thus to define a safety property as a prefix-closed property over finite traces (containing the “good traces”) – by “property” in this paper we mean a set of finite or infinite traces. Inspired by Lamport [125], Alpern and Schneider [7] define safety properties over infinite traces as ones with the property that if an infinite trace is unacceptable then there must be some finite prefix of it which is already unacceptable, in the sense that there is no acceptable infinite completion of it. Is there any relationship between these two definitions of safety? We show rather indirectly that there is, by showing that their corresponding sets of safety properties have the cardinal c of the continuum (i.e., the cardinal of \mathbb{R} , the set of real numbers), so there exists some bijective mapping between the two. Unfortunately, the existence of such a bijection is

as little informative as the existence of a bijection between the real numbers and the irrational numbers. To capture the relationship between finite- and infinite-trace safety properties in a meaningful way, we introduce a subset of finite-trace safety properties, called *persistent*, and then construct an explicit bijection between that subset and the infinite-trace safety properties. Interestingly, over finite traces there are as many safety properties as unrestricted properties (finite-traces are enumerable and $\mathcal{P}(\mathbb{N})$ is in bijection with \mathbb{R}), while over infinite traces there are c safety properties versus 2^c unrestricted properties (infinite traces are in bijection with \mathbb{R}).

It is also common to define safety properties as properties over both finite and infinite traces, the intuition for the finite traces being that of unfinished computations. For example, Lamport [126] extends the notion of infinite-trace safety properties to properties over both finite and infinite traces, while Schneider et al. [159, 82] give an alternative definition of safety over finite and infinite traces, called “execution monitoring”. One immediate technical advantage of allowing both finite and infinite traces is that one can define prefix-closed properties. We indirectly show that prefix-closeness is not a sufficient condition to define safety properties when infinite traces are also allowed, by showing that there are 2^c prefix-closed properties versus, as expected, “only” c safety properties.

Another common way to specify safety properties is as “always past” properties, that is, as properties containing only words whose finite prefixes satisfy a given property. If P is a property on finite prefixes, then we write $\Box P$ for the “always P ” safety property containing the words with prefixes in P . We show that specifying safety properties as “always past” properties is fully justified by showing that, for each of the three types of traces (finite, infinite, and both), the “always past” properties are precisely the safety properties as defined above. It is common to specify P using some logical formalism, for example past time linear temporal logic (past LTL) [129]; for example, one can specify “ a before b ” in past LTL as the formula $b \rightarrow \Diamond a$.

The problem of monitoring safety properties is also investigated in this paper. Since there are as many safety properties as real numbers, it is not unexpected that some of them can be very hard to monitor. We show that the problem of monitoring a safety property is arbitrarily hard, by showing that it reduces to deciding membership of natural

numbers to a set of natural numbers. In particular, we can associate a safety property to any degree in the arithmetic hierarchy as well as to any complexity class in the decidable universe, whose monitoring is as hard as that degree or complexity class.

From SACS: *This paper makes three novel contributions, two technical and another pedagogical. On the technical side, it first introduces the notion of a persistent safety property, which appears to be the right finite-trace correspondent of an infinite-trace safety property, and uses it to show the cardinal equivalence of the various notions of safety property encountered in the literature. Also on the technical side, it rigorously defines the problem of monitoring a safety property, and it shows that it can be arbitrarily hard. On the pedagogical side, this paper offers the first comprehensive study and uniform presentation of safety properties and of their monitoring.*

From safety: *In practice not all ($c = |\mathbb{R}|$) safety properties are meaningful, but only those ($\aleph_0 = |\mathbb{N}|$) which are specifiable using formal specification languages or logics of interest. We also investigate the problem of monitoring safety properties expressed using two common formalisms, namely regular expressions extended with complement, also called extended regular expressions (ERE), and LTL. It is known that both formalisms allow polynomial finite-trace membership checking algorithms [104, 150] if one has random access to the trace, but that both require exponential space if the trace can only be analyzed online [147, 120]. It is also known that LTL can indeed be monitored in exponential space [55] and so is claimed¹ for EREs in [147]. We show that the claim in [147] is, unfortunately, wrong, by showing that ERE monitoring requires non-elementary space. To do so, we propose for any $n \in \mathbb{N}$ a safety property P_n whose monitoring requires space non-elementary in n , as well as an ERE of size $O(n^3)$. Since the known monitoring algorithms for LTL in its full generality are asymptotically optimal, what is left to do is to consider important fragments of LTL. We focus on the “always past” fragment and give a monitor synthesis algorithm that takes formulae φ and generate monitors for them that need $O(k)$ total space and $O(|\varphi|)$ time to process each event, where k is the number of past operators in φ . This improves over the best known algorithm that needs space $O(|\varphi|)$ (and same time).*

End of intro stuff for chapter on safety

from *J.ASE 2005: Techniques for efficiently evaluating future time Linear Temporal Logic (abbreviated LTL) formulae on finite execution traces* are presented. While the standard models of LTL are infinite traces, finite traces appear naturally when testing and/or monitoring real applications that only run for limited time periods. A finite trace variant of LTL is formally defined, together with an immediate executable semantics which turns out to be quite inefficient if used directly, via rewriting, as a monitoring procedure. Then three algorithms are investigated. First, a simple synthesis algorithm for monitors based on dynamic programming is presented; despite the efficiency of the generated monitors, they unfortunately need to analyze the trace backwards, thus making them unusable in most practical situations. To circumvent this problem, two rewriting-based practical algorithms are further investigated, one using rewriting directly as a means for online monitoring, and the other using rewriting to generate automata-like monitors, called binary transition tree finite state machines (and abbreviated BTT-FSMs). Both rewriting algorithms are implemented in Maude, an executable specification language based on a very efficient implementation of term rewriting. The first rewriting algorithm essentially consists of a set of equations establishing an executable semantics of LTL, using a simple formula transforming approach. This algorithm is further improved to build automata on-the-fly via caching and reuse of rewrites (called memoization), resulting in a very efficient and small Maude program that can be used to monitor program executions. The second rewriting algorithm builds on the first one and synthesizes provably minimal BTT-FSMs from LTL formulae, which can then be used to analyze execution traces online without the need for a rewriting system. The presented work is part of an ambitious runtime verification and monitoring project at NASA Ames, called PATHEXPLORER, and demonstrates that rewriting can be a tractable and attractive means for experimenting and implementing logics for program monitoring.

next is from *J.ASE 2005*

Future time Linear Temporal Logic, abbreviated LTL, was introduced

by Pnueli in 1977 [143] (see also [128, 130]) for stating properties about reactive and concurrent systems. LTL provides temporal operators that refer to the future/remaining part of an execution trace relative to a current point of reference. The standard models of LTL are infinite execution traces, reflecting the behavior of such systems as ideally always being ready to respond to requests. Methods, such as model checking, have been developed for proving programs correct with respect to requirements specified as LTL formulae. Several systems are currently being developed that apply model checking to software systems written in Java, C and C++ [17, 56, 102, 51, 142, 72, 172, 92, 175]. However, for very large systems, there is little hope that one can actually prove correctness, and one must in those cases rely on debugging and testing. In the context of highly reliable and/or safety critical systems, one would actually want to *monitor* a program execution during operation and to determine whether it conforms to its specification. Any violation of the specification can then be used to guide the execution of the program into a safe state, either manually or automatically. In this paper we describe a collection of algorithms for monitoring program executions against LTL formulae. It is demonstrated how term rewriting, and in particular the Maude rewriting system [45], can be used to implement some of these algorithms very efficiently and conveniently.

The work presented in this paper has been started as part of, and stimulated by, the PATHEXPLORER project at NASA Ames, and in particular the Java PATHEXPLORER (JPAX) tool [93, 94] for monitoring Java programs. JPAX facilitates automated instrumentation of Java byte-code, currently using Compaq's JTREK which is not public anymore, but soon using BCEL [52]. The instrumented code emits relevant events to an observer during execution (see Figure 1.1). The observer can be running a Maude [45] process as a special case, so Maude's rewriting engine can be used to drive a temporal logic operational semantics with program execution events. The observer may run on a different computer, in which case the events are transmitted over a socket. The system is driven by a specification, stating what properties to be proved and what parts of the code to be instrumented. When the observer receives the events it dispatches these to a set of observer modules, each module performing a particular analysis that has been requested. In addition to checking temporal logic requirements, modules have also been programmed to perform error pattern analysis of multi-threaded programs, predicting deadlock and data race potentials.

Using temporal logic in testing is an idea of broad practical and theoretical

Figure 1.1: Overview of JPAX .

interest. One example is the commercial Temporal Rover and DBRover tools [57, 59], in which LTL properties are translated into code, which is then inserted at chosen positions in the program and executed whenever reached during program execution. The MaC tool [127, 115] is another example of a runtime monitoring tool. Here, Java byte-code is automatically instrumented to generate events of interest during the execution. Of special interest is the temporal logic used in MaC, which can be classified as a past time interval logic convenient for expressing monitoring properties in a succinct way. All the systems above try to discharge the program execution events as soon as possible, in order to minimize the space requirements. In contrast, a technique is proposed in [118] where the execution events are stored in an SQL database at runtime and then analyzed by means of queries after the program terminates. The PET tool, described in [81, 80, 79], uses a future time temporal logic formula to guide the execution of a program for debugging purposes. Java MultiPathExplorer [163] is a tool which checks a past time LTL safety formula against a partial order extracted online from an execution trace. POTA [160] is another partial order trace analyzer system. Java-MoP [35] is a generic logic monitoring tool encouraging “monitoring-oriented programming” as a paradigm merging specification and implementation. Complexity results for testing a finite trace against temporal formulae expressed in different temporal logics are investigated in [132]. Algorithms using alternating automata to monitor LTL properties are proposed in [63], and a specialized LTL collecting statistics along the execution trace is described in [64]. Various algorithms to generate testing automata from temporal logic formulae are discussed in [144, 141], and [71] presents a Büchi automata inspired algorithm adapted to finite trace LTL.

The major goal of this paper is to present rewriting-based algorithms for effectively and efficiently evaluating LTL formulae on finite execution traces *online*, that is, by processing each event as it arrives. An important contribution of this paper is to show how a rewriting system, such as Maude, makes it possible to experiment with monitoring logics very efficiently and elegantly, and furthermore can be used as a practical program monitoring engine. This approach allows one to formalize ideas in a framework close to standard mathematics. The presented algorithms are considered in the

context of JPAX, but they can be easily adapted and used within other monitoring frameworks. We claim that the techniques presented in this paper, even though applied to LTL, are in fact generic and can be easily applied to other logics for monitoring. For example, in [147, 161] we applied the same generic, “formula transforming”, techniques to obtain rewriting based algorithms for situations in which the logic for monitoring was replaced by extended regular expressions (regular expressions with complement).

A non-trivial application of the rewriting based techniques presented in this paper is X9, a test-case generation and monitoring environment for a software system that controls the planetary NASA rover K9. This collaborative effort is described in more detail in [13] and it will be presented in full detail elsewhere soon. The rover controller, programmed in 35,000 lines of C++, essentially executes plans, where a plan is a tree-like structure consisting of actions and sub-actions. The leaf actions control various hardware on the rover, such as for example the camera and the wheels. The execution of a plan must cause the actions to be executed in the right order and must satisfy various time constraints, also part of the plan. Actions can start and eventually either terminate successfully or fail. Plans can specify how failed sub-actions can propagate upwards.

Testing the rover controller consists of generating plans and then monitoring that the plan actions are executed in the right order and that failures are propagated correctly. X9 automatically generates plans from a “grammar” of the structure of plans, using the Java PathFinder model checker [175]. For each plan, a set of temporal formulae that an execution of the plan must satisfy is also generated. For example, a plan may state that an action a should be executed by first executing a sub-action a_1 and then a sub-action a_2 , and that the failure of any of the sub-actions should not propagate: action a should eventually succeed, regardless of whether a_1 or a_2 fails. The generated temporal formulae will state these requirements, such as for example $\Box(\text{start}(a) \rightarrow \langle \rangle \text{succeed}(a))$ saying that “it is always the case (\Box) that when action a starts, then eventually ($\langle \rangle$) it terminates successfully”, and execution traces are monitored against them.

X9 is currently being turned into a mature system to be used by the developer. It is completely automated, generating a web-page containing all the warnings found. The top-level web-page identifies all the test-cases that have failed (by violating some of the temporal properties), each linked to a web-page containing specifics such as the plan, the execution trace, and the properties that are violated. X9 has itself been tested by seeding

errors into the rover controller code. The automated monitoring relieves the programmer from manually analyzing printed execution traces. Extending the logic with real-time, as is planned in future work, is crucial for this application since plans are heavily annotated with time constraints.

In Section 1.1, based on our experience, we give a rough classification of monitoring and runtime analysis algorithms by considering three important criteria. A first criterion is whether the execution trace of the monitored or analyzed program needs to be stored or not. Storing a trace might be very useful for specific types of analysis because one could have random access to events, but storing an execution trace is an expensive operation in practice, so sometimes trace-storing algorithms may not be desirable. A second criterion regards the synchronicity of the monitor, more precisely whether the monitor is able to react as soon as the specification or the requirement has been violated. Synchronicity may often trigger running a validity checker for the logic under consideration, which is typically a very expensive task. Finally, monitoring and analysis algorithms can also be classified as “predictive” versus “exact”, where the “exact” ones monitor the observed execution trace as a flat list of events, while the predictive algorithms try to guess potential erroneous behaviors of programs that can occur under different executions. All the algorithms in this paper are exact.

This paper requires a certain amount of mathematical notions and notations, which we introduce in Section 2.1 together with Maude [45], a high-performance system supporting both membership equational logic [137] and rewriting logic [136]. The current version of Maude can do more than 3 million rewritings per second on standard PCs, and its compiled version is intended to support more than 15 million rewritings per second², so it can quite well be used as an implementation language.

Section 8.1 defines the finite trace variant of linear temporal logic that we use in the rest of the paper. We found, by carefully analyzing several practical examples, that the most appropriate assumption to make at the end of the trace is that it is stationary in the last state. Then we define the semantics of the temporal operators using their usual meaning in infinite trace LTL, where the finite trace is infinitely extended by repeating the last state. Another option would be to consider that all atomic predicates are false or true in the state following the last one, but this would be problematic when inter-dependent predicates are involved, such as “gate-up” and “gate-down”.

²Personal communication by José Meseguer.

In previous work we described a technique which synthesizes efficient dynamic programming algorithms for checking LTL formulae on finite execution traces [149]. Even though this algorithm is not dependent on rewriting (but it could be easily implemented in Maude by rewriting as we did with its dual variant for past time LTL [87, 35]), for the sake of completeness we present it in some detail in Section 8.2. This algorithm evaluates a formula bottom-up for each point in the trace, going backwards from the final state towards the initial state. Unfortunately, despite its linear complexity, this algorithm cannot be used online because it is both asynchronous and trace-storing. In [88, 89, 87] we dualize this technique and apply it to past time LTL, in which case the trace more naturally can be examined in a forwards direction synchronously.

Section 8.3 presents our first practical rewriting-based algorithm, which can directly monitor an LTL formula. This algorithm originates in [83, 149] and it was partially presented at the Automated Software Engineering conference [85]. The algorithm is expressed as a set of equations establishing an executable semantics of LTL using a simple formula transforming approach. The idea is to rewrite or transform an LTL monitoring requirement formula φ when an event e is received, to a formula $\varphi\{e\}$, which represents the new requirement that the monitored system should fulfill for the remaining part of the trace. This way, the LTL formula to monitor “evolves” into other LTL formulae by subsequent transformations. We show, however, that the size of the evolving formula is in the worst-case exponentially bounded by the size of the original LTL formula, and also that an exponential space explosion cannot be avoided in certain unfortunate cases. The efficiency of this rewriting algorithm can be improved by almost an order of magnitude by caching and reusing rewrites (a.k.a. “memoization”), which is supported by Maude. This algorithm is often synchronous, though there are situations in which it misses reporting a violation at the exact event when it occurs. The violation is, however, detected at a subsequent event. This algorithm can be relatively easily transformed into a synchronous one if one is willing to pay the price of running a validity checker, like the one presented in Subsection 8.4.1, after processing each event. The practical result of Section 8.3 is a very efficient and small Maude program that can be used to monitor program executions. The decision to use Maude has made it very easy to experiment with logics and algorithms in monitoring.

We finally present an alternative solution to monitoring LTL in Section 8.4, where a rewriting-based algorithm is used to *generate* an optimal special

observer from an LTL formula. By optimality is meant everything one may expect, such as minimal number of states, forwards traversal of execution traces, synchronicity, efficiency, but also less standard optimality features, such as transiting from one state to another with a minimum amount of computation. In order to effectively do this we introduce the notion of *binary transition tree* (BTT), as a generalization of binary decision diagrams (BDD) [29], whose purpose is to provide an *optimal order* in which state predicates need to be evaluated to decide the next state. The motivation for this is that in practical applications evaluating a state predicate is a time consuming task, such as for example to check whether a vector is sorted. The associated finite state machines are called *binary transition tree finite state machines* (BTT-FSM). BTT-FSMs can be used to analyze execution traces without the need for a rewriting system, and can hence be used by observers written in traditional programming languages. The BTT-FSM generator, which includes a validity checker, is also implemented in Maude and has about 200 lines of code in total.

1.1 A Taxonomy of Runtime Analysis Techniques

A *runtime analysis technique* is regarded in a broad sense in this section; it can be a method or a concrete algorithm that analyzes the execution trace of a running program and concludes a certain property about that program. Runtime analysis algorithms can be arbitrarily complex, depending upon the kind of properties to be monitored or analyzed. Based on our experience with current procedures implemented in JPaX, in this section we make an attempt to classify runtime analysis techniques. The three criteria below are intended to be neither exhaustive nor always applicable, but we found them quite useful in practice. They are not specific to any particular logic or approach, so we present them before we introduce our logic and algorithms. In fact, this taxonomy will allow us to appropriately discuss the benefits and drawbacks of our algorithms presented in the rest of the paper.

1.1.1 Trace Storing versus Non-Storing Algorithms

As events are received from the monitored system, a runtime analysis algorithm typically maintains a state which allows it to reason about the monitored execution trace. Ideally, the amount of information needed to be stored by the monitor in its state depends only upon the property to be

monitored and *not* upon the number of already processed events. This is desired because, due to the huge amount of events that can be generated during a monitoring session, one would want one's monitoring algorithms to work in linear time with the number of events processed.

There are, however, situations where it is not possible or practically feasible to use storage whose size is a function of only the monitoring requirement. One example is that of monitoring *extended regular expressions* (ERE), i.e., regular expressions enriched with a complement operator. As shown by the success of scripting languages like PERL or PYTHON, software developers tend to understand and like regular expressions and feel comfortable to describe patterns using those, so ERE is a good candidate formalism to specify monitoring requirements (we limit ourselves to only patterns described via temporal logics in this paper though).

It is however known that ERE to automata translation algorithms suffer from a non-elementary state explosion problem, because a complement operation requires nondeterministic-to-deterministic automata conversions, which yield exponential blowups in the number of states. Since complement operations can be nested, generating automata from EREs may often not be feasible in practice. Fortunately, there are algorithms which avoid this state explosion problem, at the expense of having to store the execution trace and then, at the end of the monitoring session, to analyze it by traversing it forwards and backwards many times. The interested reader is referred to [104] for a $O(n^3m)$ dynamic programming algorithm (n is the length of the execution trace and m is the size of the ERE), and to [177, 121] for $O(n^2m)$ non-trivial algorithms.

Based on these observations, we propose a first criterion to classify monitoring algorithms, namely on whether they *store or do not store the execution trace*. In the case of EREs, trace storing algorithms are polynomial in the size of the trace and linear in the ERE requirement, while the non-storing ones are linear in the size of the trace and highly exponential in the size of the requirement. In this paper we show that trace storing algorithms for linear temporal logic can be linear in both the trace and the requirement (see Section 8.2), while trace non-storing ones are linear in the size of the trace but simply exponential in the size of the requirement.

Trace non-storing algorithms are apparently preferred, but, however, their size can be so big that it could make their use unamenable in certain important situations. One should carefully analyze the trade-offs in order to make the best choice in a particular situation.

1.1.2 Synchronous versus Asynchronous Monitoring

There are many safety critical applications in which one would want to report a violation of a requirement as soon as possible, and to not allow the monitored program to take any further action once a requirement is violated. We call this desired functionality *synchronous monitoring*. Otherwise, if a violation can only be detected after the monitored program executes several more steps or after it is stopped and its entire execution trace is needed to perform the analysis, then we call it *asynchronous monitoring*.

The dynamic programming algorithm presented in Section 8.2 is *not* synchronous, because one can detect a violation only after the program is stopped and its execution trace is available for backwards traversal. The algorithm presented in Section 8.3 is also asynchronous in general because there are universally false formulae which are detected so only at the end of an execution trace or only after several other execution steps. Consider, for example, that one monitors the finite trace LTL formula $\neg \langle \rangle (\Box A \vee \Box \neg A)$, which is false because at the end of any execution trace A either holds or not, or the formula $\Box \Box A \wedge \Box \Box \neg A$, which is also false but will be detected so only after two more events. However, the rewriting algorithm in Section 8.3 is synchronous in many practical situations. The algorithm in Section 8.4 is always synchronous, though one should be aware of the fact that its size may become a problem on large formulae.

In order for an LTL monitor to be synchronous, it needs to implement a validity checker for finite trace LTL, such as the one in Subsection 8.4.1 (Figure 8.2), and call it on the current formula after each event is processed. Checking validity of a finite trace LTL formula is very expensive (we are not aware of any theoretical result stating its exact complexity, but we believe that it is PSPACE-complete, like for standard infinite trace LTL [167]). We are currently considering providing a fully synchronous LTL monitoring module within JPAX, at the expense of calling a validity checker after each event, and let the user of the system choose either synchronous or asynchronous monitoring.

There are, however, many practical LTL formulae for which violation can be detected synchronously by the formula transforming rewriting-based algorithm presented in Section 8.3. Consider for example the sample formula of this paper, $\Box (\text{green} \rightarrow \neg \text{red} \cup \text{yellow})$, which is violated if and only if a red event is observed after a green one. The monitoring requirement of our algorithm, which initially is the formula itself, will not be changed unless a green event is received, in which case it will change to

$(\text{!red} \cup \text{yellow}) \wedge [](\text{green} \rightarrow \text{!red} \cup \text{yellow})$. A yellow event will turn it back into the initial formula, a green event will keep it unchanged, but a red event will turn it into **false**. If this is the case, then the monitor declares the formula violated and appropriate actions can be taken. Notice that the violation was detected *exactly* when it occurred. A very interesting, practical and challenging problem is to find criteria that say when a formula can be synchronously monitored without the use of a validity checker.

1.1.3 Predictive versus Exact Analysis

An increasingly important class of runtime analysis algorithms are concerned with *predicting* anomalies in programs from *successful* observed executions. One such algorithm can be easily obtained by slightly modifying the *wait-for-graph* algorithm, which is typically used to *detect* when a system is in a deadlock state, to make it predict deadlocks. One way to do this is to *not* remove synchronization objects from the wait-for-graph when threads/processes release them. Then even though a system is not deadlock, a warning can be reported to users if a cycle is found in the wait-for-graph, because that represents a *potential* of a deadlock.

Another algorithm falling into the same category is Eraser [157], a datarace prediction procedure. For each shared memory region, Eraser maintains a set of *active locks* which protect it, which is intersected with the set of locks held by any accessing thread. If the set of active locks ever becomes empty then a warning is issued to the user, with the meaning that a potential unprotected access can take place. Both the deadlock and the datarace predictive algorithms are very successful in practice because they scale well and find many of the errors they are designed for. We have also implemented improved versions of these algorithms in Java PathExplorer.

We are currently also investigating predictive analysis of safety properties expressed using past time temporal logic, and a prototype system called Java MultiPathExplorer is being implemented [163]. The main idea here is to *instrument* Java classes to emit events timestamped by vector clocks [62], thus enabling the observer to extract a *partial order* reflecting the causal dependency on the memory accesses of the multithreaded program. If any linearization of that inferred partial order leads to a violation of the safety property then a warning is generated to the user, with the meaning that there can be executions of the multithreaded program, including the current one, which violate the requirements.

In this paper we restrict ourselves to only *exact* analysis of execution

traces. That means that the events in the trace are supposed to have occurred exactly in the received order (this can be easily enforced by maintaining a logical clock, then timestamping each event with the current clock, and then delivering the messages in increasing order of timestamps), and that we only check whether that particular order violates the monitoring requirements or not. Techniques for predicting future time LTL violations will be investigated elsewhere soon.

Although the taxonomy discussed in this section is intended to only be applied to tools, the problem domain may also admit a similar taxonomy. While such a taxonomy seems to be hard to accomplish in general, it would certainly be very useful because it would allow one to choose the proper runtime analysis technique for a given system and set of properties. However, like this paper shows, it is often the case that one can choose among several types of runtime analysis techniques for a given problem domain.

Chapter 2

Background, Preliminaries, Notations

Add some structure to this chapter

from RV03 and RTA03

We let \mathbb{N} denote the set of natural numbers including 0 but excluding the infinity symbol ∞ and let \mathbb{N}_∞ denote the set $\mathbb{N} \cup \{\infty\}$. We also let \mathbb{Q} denote the set of rational numbers and \mathbb{R} the set of real numbers; as for natural numbers, the “ ∞ ” subscript can also be added to \mathbb{Q} and \mathbb{R} for the corresponding extensions of these sets. \mathbb{Q}^+ and \mathbb{R}^+ denote the sets of strictly positive (0 not included) rational and real numbers, respectively.

We fix a set Σ of elements called *events* or *states*. We call words in Σ^* *finite traces* and those in Σ^ω *infinite traces*. If $u \in \Sigma^* \cup \Sigma^\omega$ then u_i is the i -th state or event that appears in u . We call *finite-trace properties* sets $P \subseteq \Sigma^*$ of finite traces, *infinite-trace properties* sets $P \subseteq \Sigma^\omega$ of infinite traces, and just *properties* sets $P \subseteq \Sigma^* \cup \Sigma^\omega$ of finite or infinite traces. If the finite or infinite aspect of traces is understood from context, then we may call any of the types or properties above just *properties*. We may write $P(w)$ for a property P and a (finite or infinite) trace w whenever $w \in P$. Traces and properties are more commonly called *words* and *languages*, respectively, in the literature; we prefer to call them traces and properties to better reflect the intuition that our target application is monitoring and system observance, not formal languages. We take, however, the liberty to also call them words and languages whenever that terminology seems more appropriate.

In some cases states can be simply identified with their names, or labels, and specifications of properties on traces may just refer to those labels. For example, the regular expression $(s_1 \cdot s_2)^*$ specifies all those finite traces starting with state s_1 and in which states s_1 and s_2 alternate. In other cases, one can think of states as sets of atomic predicates, that is, predicates that hold in those states: if s is a state and a is an atomic predicate, then we say that $a(s)$ is true iff a “holds” in s ; thus, if all it matters with respect to states is which predicates hold and which do not hold in each state, then states can be faithfully identified with sets of predicates. We prefer to stay loose with respect to what “holds” means, because, depending on the context, it can mean anything. In conventional software situations, atomic predicates can be: boolean expressions over variables of the program, their satisfaction being decided by evaluating them in the current state of the program; or whether a function is being called or returned from; or whether a particular variable is being written to; or whether a particular lock is being held by a particular thread; and so on. In the presence of atomic predicates, specifications of properties on traces typically only refer to the atomic predicates. For example, the property “always a before b ”, that is, those traces containing no state in which b holds that is not preceded by some state in which a holds (for example, a can stand for “authentication” and b for “resource access”), can be expressed in LTL as the formula $\Box(b \rightarrow \Diamond a)$.

Let us recall some basic notions and notations from formal languages, temporarily using the consecrated terminology of “words” and “languages” instead of traces and properties. For an alphabet Σ , let \mathcal{L}_Σ be the set of languages over Σ , i.e., the powerset $\mathcal{P}(\Sigma^*)$. By abuse of language and notation, let \emptyset be the empty language $\{\}$ and ϵ the language containing only the empty word, $\{\epsilon\}$. If $L_1, L_2 \in \mathcal{L}_\Sigma$ then $L_1 \cdot L_2$ is the language $\{\alpha_1\alpha_2 \mid \alpha_1 \in L_1 \text{ and } \alpha_2 \in L_2\}$. Note that $L \cdot \emptyset = \emptyset \cdot L = \emptyset$ and $L \cdot \epsilon = \epsilon \cdot L = L$. If $L \in \mathcal{L}_\Sigma$ then L^* is $\{\alpha_1\alpha_2 \cdots \alpha_n \mid n \geq 0 \text{ and } \alpha_1, \alpha_2, \dots, \alpha_n \in L\}$ and $\neg L$ is $\Sigma^* - L$.

We next recall some notions related to cardinality. If A is any set, we let $|A|$ denote the *cardinal* of A , which expresses the size of A . When A is finite, $|A|$ is precisely the number of elements of A and we call it a *finite cardinal*. Infinite sets can have different cardinals, called *transfinite* or even *infinite*. For example, natural numbers \mathbb{N} have the cardinal \aleph_0 (pronounced “aleph zero”) and real numbers \mathbb{R} have the cardinal c , also called the *cardinal of the continuum*. Two sets A and B are said to have the same cardinal, written $|A| = |B|$, iff there is some bijective mapping between the two. We

write $|A| \leq |B|$ iff there is some injective mapping from A to B .

The famous *Cantor-Bernstein-Schroeder theorem* states that if $|A| \leq |B|$ and $|B| \leq |A|$ then $|A| = |B|$. In other words, to show that there is some bijection between sets A and B , it suffices to find an injection from A to B and an injection from B to A . The two injections need not be bijections. For example, the inclusion of the interval $(0, 1)$ in \mathbb{R}^+ is obviously an injection, so $|(0, 1)| \leq |\mathbb{R}^+|$. On the other hand, the function $x \mapsto x/(2x + 1)$ from \mathbb{R}^+ to $(0, 1)$ (in fact its codomain is the interval $(0, 1/2)$) is also injective, so $|\mathbb{R}^+| \leq |(0, 1)|$. Neither of the two injective functions is bijective, yet by the Cantor-Bernstein-Schroeder theorem there is some bijection between $(0, 1)$ and \mathbb{R}^+ , that is, $|(0, 1)| = |\mathbb{R}^+|$. We will use this theorem to relate the various types of safety properties; for example, we will show that there is an injective function from safety properties over finite traces to safety properties over infinite traces and another injective function in the opposite direction. Unfortunately, the Cantor-Bernstein-Schroeder theorem is existential: it only says that some bijection exists between the two sets, but it does not give us an explicit bijection. Since the visualization of a concrete bijection between different sets of safety properties can be very meaningful, we will avoid using the Cantor-Bernstein-Schroeder theorem when we can find an explicit bijection between two sets of safety properties.

If A is a set of cardinal α , then 2^α is the cardinal of $\mathcal{P}(A)$, the power set of A (the set of subsets of A). It is known that $2^{\aleph_0} = c$, that is, there are as many sets of natural numbers as real numbers. The famous, still unanswered *continuum hypothesis*, states that there is no set whose size is strictly between \aleph_0 and c ; more generally, it states that, for any transfinite cardinal α , there is no proper cardinal between α and 2^α . If A and B are infinite sets, then $|A| + |B|$ and $|A| \cdot |B|$ are the cardinals of the sets $A \cup B$ and $A \times B$, respectively. An important property of transfinite cardinals is that of *absorption* – the larger cardinal absorbs the smaller one: if α and β are transfinite cardinals such that $\alpha \leq \beta$, then $\alpha + \beta = \alpha \cdot \beta = \beta$; in particular, $c \cdot 2^c = 2^c$. Besides sets of natural numbers, there are several other important sets that have cardinal c : streams (i.e., infinite sequences) of Booleans, streams of reals, non-empty closed or open intervals of reals, as well as the sets of all open or closed sets of reals, respectively (Exercise 1).

For our purposes, if Σ is an enumerable set of states, then Σ^* is also enumerable, so it has cardinal \aleph_0 . Also, if $|\Sigma| \leq c$, in particular if it is finite, then Σ^ω has the cardinal c , because it is equivalent to streams of states. We can then immediately infer that the set of finite-trace properties over Σ has

cardinal $2^{\aleph_0} = c$, while the set of infinite-trace properties has cardinal 2^c .

end from RV03 and RTA03

Exercises

Exercise 1 *Show that each of the following sets have cardinal c : streams (i.e., infinite sequences) of Booleans; streams of natural numbers; streams of real numbers; closed intervals of real numbers; open intervals of real numbers; closed sets of real numbers; open sets of real numbers.*

from J.ASE'05

2.1 Preliminaries

In this section we recall notions and notations that will be used in the paper, including membership equational logic, term rewriting, Maude notation, and (infinite trace) linear temporal logics.

2.1.1 Membership Equational Logic

Membership equational logic (MEL) extends many- and order-sorted equational logic by allowing memberships of terms to sorts in addition to the usual equational sentences. We only recall those MEL notions which are necessary for understanding this paper; the interested reader is referred to [137, 28] for a comprehensive exposition of MEL.

Basic Definition

A *many-kinded algebraic signature* (K, Σ) consists of a set K and a $(K^* \times K)$ -indexed set $\Sigma = \{\Sigma_{k_1 k_2 \dots k_n, k} \mid k_1, k_2, \dots, k_n, k \in K\}$ of operations, where an operation $\sigma \in \Sigma_{k_1 k_2 \dots k_n, k}$ is written $\sigma : k_1 k_2 \dots k_n \rightarrow k$. A *membership signature* Ω is a triple (K, Σ, π) where K is a set of *kinds*, Σ is a K -kinded algebraic signature, and $\pi : S \rightarrow K$ is a function that assigns to each element in its domain, called a *sort*, a kind. Therefore, sorts are grouped according to kinds and operations are defined on kinds. For simplicity, we will call a “membership signature” just a “signature” whenever there is no confusion.

For a *many-kinded signature* (K, Σ) , a Σ -algebra A consists of a K -indexed set $\{A_k \mid k \in K\}$ together with interpretations of operations $\sigma : k_1 k_2 \dots k_n \rightarrow k$ into functions $A_\sigma : A_{k_1} \times A_{k_2} \times \dots \times A_{k_n} \rightarrow A_k$. For any given signature $\Omega = (K, \Sigma, \pi)$, an Ω -membership algebra A is a Σ -algebra together with a set $A_s \subseteq A_{\pi(s)}$ for each sort $s \in S$. A particular algebra, called *term algebra*, is of special interest. Given a K -kinded signature Σ and a K -indexed set of *variables* X , let $T_\Sigma(X)$ be the algebra of Σ -terms over variables in X extending X iteratively as follows: if $\sigma : k_1 k_2 \dots k_n \rightarrow k$ and $t_1 \in T_{\Sigma, k_1}(X)$, $t_2 \in T_{\Sigma, k_2}(X)$, ..., $t_n \in T_{\Sigma, k_n}(X)$, then $\sigma(t_1, t_2, \dots, t_n) \in T_{\Sigma, k}(X)$.

Given a signature Ω and a K -indexed set of variables X , an *atomic* (Ω, X) -equation has the form $t = t'$, where $t, t' \in T_{\Sigma, k}(X)$, and an *atomic* (Ω, X) -membership has the form $t : s$, where s is a sort and $t \in T_{\Sigma, \pi(s)}(X)$. An Ω -sentence in MEL has the form $(\forall X) a \text{ if } a_1 \wedge \dots \wedge a_n$, where a, a_1, \dots, a_n are atomic (Ω, X) -equations or (Ω, X) -memberships, and $\{a_1, \dots, a_n\}$ is a set (no duplications). If $n = 0$, then the Ω -sentence is called *unconditional* and written $(\forall X) a$. Equations are called *rewriting rules* when they are used only from left to right, as it will happen in this paper.

Given an Ω -algebra A and a K -kinded map $\theta : X \rightarrow A$, then $A, \theta \models_\Omega t = t'$ iff $\theta(t) = \theta(t')$, and $A, \theta \models_\Omega t : s$ iff $\theta(t) \in A_s$. A satisfies $(\forall X) a \text{ if } a_1 \wedge \dots \wedge a_n$, written $A \models_\Omega (\forall X) a \text{ if } a_1 \wedge \dots \wedge a_n$, iff for each $\theta : X \rightarrow A$, if $A, \theta \models_\Omega a_1$ and ... and $A, \theta \models_\Omega a_n$, then $A, \theta \models_\Omega a$.

An Ω -specification (or Ω -theory) $T = (\Omega, E)$ in MEL consists of a signature Ω and a set E of Ω -sentences. An Ω -algebra A satisfies (or is a model of) $T = (\Omega, E)$, written $A \models T$, iff it satisfies each sentence in E .

Inference Rules

MEL admits complete deduction (see [137], where the rule of congruence is stated in a somewhat different but equivalent way). In the congruence rule below, $\sigma \in \Sigma_{k_1 \dots k_i, k}$, W is a set of variables $w_1 : k_1, \dots, w_{i-1} : k_{i-1}, w_{i+1} : k_{i+1}, \dots, w_n : k_n$, and $\sigma(W, t)$ is a shorthand for the term

$\sigma(w_1, \dots, w_{i-1}, t, w_{i+1}, \dots, w_n)$:

- (1) Reflexivity :
$$\frac{}{E \vdash_{\Omega} (\forall X) t = t}$$
- (2) Symmetry :
$$\frac{E \vdash_{\Omega} (\forall X) t = t'}{E \vdash_{\Omega} (\forall X) t' = t}$$
- (3) Transitivity :
$$\frac{E \vdash_{\Omega} (\forall X) t = t', E \vdash_{\Omega} (\forall X) t' = t''}{E \vdash_{\Omega} (\forall X) t = t''}$$
- (4) Congruence :
$$\frac{E \vdash_{\Omega} (\forall X) t = t'}{E \vdash_{\Omega} (\forall X, W) \sigma(W, t) = \sigma(W, t'), \text{ for each } \sigma \in \Sigma}$$
- (5) Membership :
$$\frac{E \vdash_{\Omega} (\forall X) t = t', E \vdash_{\Omega} (\forall X) t : s}{E \vdash_{\Omega} (\forall X) t' : s}$$
- (6) Modus Ponens :
$$\left\{ \begin{array}{l} \text{Given a sentence in } E \\ (\forall Y) t = t' \text{ if } t_1 = t'_1 \wedge \dots \wedge t_n = t'_n \wedge w_1 : s_1 \wedge \dots \wedge w_m : s_m \\ \text{(resp. } (\forall Y) t : s \text{ if } t_1 = t'_1 \wedge \dots \wedge t_n = t'_n \wedge w_1 : s_1 \wedge \dots \wedge w_m : s_m) \\ \text{and } \theta : Y \rightarrow T_{\Sigma}(X) \text{ s.t. for all } i \in \{1, \dots, n\} \text{ and } j \in \{1, \dots, m\} \\ E \vdash_{\Omega} (\forall X) \theta(t_i) = \theta(t'_i), E \vdash_{\Omega} (\forall X) \theta(w_j) : s_j \\ \hline E \vdash_{\Omega} (\forall X) \theta(t) = \theta(t') \quad (\text{resp. } E \vdash_{\Omega} (\forall X) \theta(t) : s) \end{array} \right.$$

The rules above can therefore prove any unconditional equation or membership that is true in all membership algebras satisfying E . In order to derive conditional statements, we will therefore consider the standard technique adapting the “deduction theorem” to equational logics, namely deriving the conclusion of the sentence after adding the condition as an axiom; in order for this procedure to be correct, the variables used in the conclusion need to be first transformed into constants. All variables can be transformed into constants, so we only consider the following simplified rules:

- (7) Theorem of Constants :
$$\frac{E \vdash_{\Omega \cup X} (\forall \emptyset) a \text{ if } a_1 \wedge \dots \wedge a_n}{E \vdash_{\Omega} (\forall X) a \text{ if } a_1 \wedge \dots \wedge a_n}$$
- (8) Implication Elimination :
$$\frac{E \cup \{a_1, \dots, a_n\} \vdash_{\Omega} (\forall \emptyset) a}{E \vdash_{\Omega} (\forall \emptyset) a \text{ if } a_1 \wedge \dots \wedge a_n}$$

Theorem 1 (from [137]) *With the notation above, $E \models_{\Omega} (\forall X) a$ if $a_1 \wedge \dots \wedge a_n$ if and only if $E \vdash_{\Omega} (\forall X) a$ if $a_1 \wedge \dots \wedge a_n$. Moreover, any statement can be proved by first applying rule (7), then (8), and then a series of rules (1) to (6).*

This theorem is used within the correctness proof of the monitoring algorithm in Section 8.3.

Initial Semantics and Induction

MEL specifications are often intended to allow only a restricted class of models (or algebras). For example, a specification of natural numbers defined using the Peano equational axioms would have many “undesired” models, such as models in which the addition operation is not commutative, or models in which, for example, $10=0$. We restrict the class of models of a MEL specification only to those which are *initial*, that is, those which obey the *no junk no confusion* principle; therefore, our specifications have *initial semantics* [76] in this paper. Intuitively, that means that only those models are allowed in which all elements can be “constructed” from smaller elements and in which no terms which cannot be proved equal are interpreted to the same elements.

By reducing the class of models, one can enlarge the class of sound inference rules. A major benefit one gets under initial semantics is that *proofs by induction become valid*. Since the proof of correctness for the main algorithm in this paper is done by induction on the structure of the temporal formula to monitor, it is important for the reader to be aware that the specifications presented from now on have initial semantics.

Syntactic Sugar Conventions

To make specifications easier to read, the following syntactic sugar conventions are widely accepted:

Subsorts. Given sorts s, s' with $\pi(s) = \pi(s') = k$, the declaration $s < s'$ is syntactic sugar for the conditional membership $(\forall x : k) x : s' \text{ if } x : s$.

Operations. If $\sigma \in \Omega_{k_1 \dots k_n, k}$ and $s_1, \dots, s_n, s \in S$ with $\pi(s_1) = k_1, \dots, \pi(s_n) = k_n, \pi(s) = k$, then the declaration $\sigma : s_1 \cdots s_n \rightarrow s$ is syntactic sugar for $(\forall x_1 : k_1, \dots, x_n : k_n) \sigma(x_1, \dots, x_n) : s \text{ if } x_1 : s_1 \wedge \dots \wedge x_n : s_n$.

Variables. $(\forall x : s, X) \text{ } a \text{ if } a_1 \wedge \dots \wedge a_n$ is syntactic sugar for the Ω -sentence $(\forall x : \pi(s), X) \text{ } a \text{ if } a_1 \wedge \dots \wedge a_n \wedge x : s$. With this, the operation declaration $\sigma : s_1 \cdots s_n \rightarrow s$ above is equivalent to $(\forall x_1 : s_1, \dots, x_n : s_n) \sigma(x_1, \dots, x_n) : s$.

2.1.2 Maude

Maude [45] is a freely distributed high-performance system in the OBJ [77] algebraic specification family, supporting both rewriting logic [136] and membership equational logic [137]. Because of its efficient rewriting engine, able to execute 3 million rewriting steps per second on standard PCs, and because of its metalanguage features, Maude turns out to be an excellent tool to create executable environments for various logics, models of computation, theorem provers, and even programming languages. We were delighted to notice how easily we could implement and efficiently validate our algorithms for testing LTL formulae on finite event traces in Maude, admittedly a tedious task in C++ or Java, and hence decided to use Maude at least for the prototyping stage of our runtime check algorithms.

We informally describe some of Maude’s features via examples in this section, referring the interested reader to its manual [45] for more details. The examples discussed in this subsection are not random. On the one hand they show all the major features of Maude that we need, while on the other hand they are part of our current JPaX implementation; several references to them will be made later in the paper. Maude supports modularization in the OBJ style. There are various kinds of modules, but we use only functional modules which follow the pattern “`fmod <name> is <body> endfm`”, and which have initial semantics. The body of a functional module consists of a collection of declarations, of which we are using importation, sorts, subsorts, operations, variables and equations, usually in this order.

Defining Logics for Monitoring

In the following we introduce some modules that we think are general enough to be used within any logical environment for program monitoring that one would want to implement by rewriting. The first one simply defines atomic propositions as an abstract data type having one sort, `Atom`, and no operations or constraints:

```
fmod ATOM is
  sort Atom .
```



```
endfm
```

The actual names of atomic propositions will be automatically generated in another module that extends `ATOM`, as constants of sort `Atom`. These will be generated by the observer at the initialization of monitoring, from the actual properties that one wants to monitor.

An important concept in program monitoring is that of an (abstract) execution trace, which consists of a finite list of events. We abstract a single event by a list of atoms, those that hold after the action that generated the event took place. The values of the atomic propositions are updated by the observer according to the actual state of the executing program and then sent to Maude as a term of sort `Event` (more details regarding the communication between the running program and Maude will be given later):

```
fmod TRACE is
  protecting ATOM .
  sorts Event Event* Trace .
  subsorts Atom < Event < Event* < Trace .
  op empty : -> Event .
  op _ : Event Event -> Event [assoc comm id: empty prec 23] .
  var A : Atom .
  eq A A = A .
  op _* : Event -> Event* .
  op _ , _ : Event Trace -> Trace [prec 25] .
endfm
```

The statement `protecting ATOM` imports the module `ATOM` without changing its initial semantics. The above is a compact way to use *mix-fix*¹ and order-sorted notation to define an abstract data type of traces: a trace is a comma separated list of events, where an event is itself a *set* of atoms. The `subsorts` declaration declares `Atom` to be a subsort of `Event`, which in turn is a subsort of `Event*` which is a subsort of `Trace`. Since elements of a subsort can occur as elements of a supersort without explicit lifting, we have as a consequence that a single event is also a trace, consisting of one event. Likewise, an atomic proposition can occur as an event, containing only this atomic proposition.

Operations can have attributes, such as associativity (A), commutativity (C), identity (I) as well as precedences, which are written between square brackets. When a binary operation is declared using the attributes A, C, and/or I, Maude uses built-in efficient specialized algorithms for matching

¹Underscores are places for arguments.

and rewriting. However, semantically speaking, the A, C, and/or I attributes can be replaced by there corresponding equations. The attribute `prec` gives a precedence to an operator², thus eliminating the need for most parentheses. Notice the special sort `Event*` which stays for terminal events, i.e., events that occur at the end of traces. Any event can potentially occur at the end of a trace. It is often the case that ending events are treated differently, like in the case of finite trace linear temporal logic; for this reason, we have introduced the operation `_*` which marks an event as terminal.

An event is defined as a set of atoms which should in fact be thought of as the set of all those atoms which “hold” in the new state of the event emitting program. Note the idempotency equation “`eq A A = A`”, which ensures that an event is indeed a set. On the other hand, a trace is a an ordered list of events which can potentially have repetitions of events. For example, the event “ $x = 5$ ” can occur several times during the execution of a program. Note that there is no need and consequently no definition of an empty trace.

Syntax and semantics are basic requirements to any logic. The following module introduces what we believe are the basic ingredients of monitoring logics, i.e., logics used for specifying monitoring requirements:

```
fmod LOGICS-BASIC is
  protecting TRACE .
  sort Formula .
  subsort Atom < Formula .
  ops true false : -> Formula .
  op [_] : Formula -> Bool .
  eq [true]  = true .
  eq [false] = false .

  var A : Atom .
  var T : Trace .
  var E : Event .
  var E* : Event* .
  op _{[_]} : Formula Event* -> Formula [prec 10] .
  eq true{E*}  = true .
  eq false{E*} = false .
  eq A{A E} = true .
  eq A{E} = false [owise] .
  eq A{E *} = A{E} .
```

²The lower the precedence number, the tighter the binding.

```

op _|=_ : Trace Formula -> Bool [prec 30] .
eq T |= true  = true .
eq T |= false = false .
eq E   |= A   = [A{E}] .
eq E,T |= A   = E |= A .
endfm

```

The first block of declarations introduces the sort `Formula` which can be thought of as a generic sort for any well-formed formula in any logic. There are two designated formulae, namely `true` and `false`, with the obvious meaning in any monitoring logic. The sort `Bool` is built-in in Maude together with two constants `true` and `false`, which are different from those of sort `Formula`, and a generic operator `if_then_else-fi`. The “interpretation” operator `[_]` maps a formula to a Boolean value. Each logic implemented on top of LOGICS-BASIC is free to define it appropriately; here we only give the obvious mappings of `true` and `false` of `Formula` into `true` and `false` of `Bool`.

The second block defines the operation `_{}_` which takes a formula and an event and yields another formula. The intuition for this operation is that it “evaluates” the formula in the new state and produces a proof obligation as another formula for the subsequent events. If the returned formula is `true` or `false` then it means that the formula was satisfied or violated, regardless of the rest of the execution trace; in this case, a message can be returned by the observer. Each logic will further complete the definition of this operator. Note that the equation “`eq A{A E} = true`” speculates Maude’s capability of performing matching modulo associativity, commutativity and identity (the attributes of the *set* concatenation on events); it basically says that `A{E}` is `true` if `E` contains the atom `A`. The next equation contains the attribute `[owise]`, stating that it should be applied only if any other equation fails to apply at a particular position.

Finally, the satisfaction relation is defined. Two obvious equations deal with the formulae `true` and `false`. The last two equations state that a trace satisfies an atomic proposition `A` if evaluating that atomic proposition `A` on the first event in the trace yields `true`. The remaining elements in the trace do not matter because `A` is a simple atom, so it refers to only the current state.

Defining Propositional Calculus

A rewriting decision procedure for propositional calculus due to Hsiang [106] is adapted and presented. It provides the usual connectives `_/_` and `_/_`.

(and), $_++_$ (exclusive or), $_\\/_$ (or), $\!_$ (negation), $__$ (implication), and $_<->_$ (equivalence). The procedure reduces tautological formulae to the constant `true` and all the others to some canonical form modulo associativity and commutativity. An unusual aspect of this procedure is that a canonical form consists of an exclusive or of conjunctions. In fact, this choice of basic operators corresponds to regarding propositional calculus as a Boolean ring rather than as a Boolean algebra. A major advantage of this choice is that normal forms are unique modulo associativity and commutativity. Even if propositional calculus is very basic to almost any logical environment, we decided to keep it as a separate logic instead of being part of the logic infrastructure of JPAX. One reason for this decision is that its operational semantics could be in conflict with other logics, for example ones in which conjunctive normal forms are desired.

An OBJ3 code for this procedure appeared in [77]. Below we give its obvious translation to Maude together with its finite trace semantics, noticing that Hsiang [106] showed that this rewriting system modulo associativity and commutativity is Church-Rosser and terminates. The Maude team was probably also inspired by this procedure, since the builtin `BOOL` module is very similar.

```
fmod PROP-CALC is
  extending LOGICS-BASIC .
*** Constructors ***
  op _/__ : Formula Formula -> Formula [assoc comm prec 15] .
  op _++_ : Formula Formula -> Formula [assoc comm prec 17] .
  vars X Y Z : Formula .
  eq true /\ X = X .
  eq false /\ X = false .
  eq X /\ X = X .
  eq false ++ X = X .
  eq X ++ X = false .
  eq X /\ (Y ++ Z) = X /\ Y ++ X /\ Z .
*** Derived operators ***
  op _\\/_ : Formula Formula -> Formula [assoc prec 19] .
  op !_ : Formula -> Formula [prec 13] .
  op _->_ : Formula Formula -> Formula [prec 21] .
  op _<->_ : Formula Formula -> Formula [prec 23] .
  eq X \\ Y = X /\ Y ++ X ++ Y .
  eq ! X = true ++ X .
  eq X -> Y = true ++ X ++ X /\ Y .
  eq X <-> Y = true ++ X ++ Y .
*** Finite trace semantics
```

```

var T : Trace .
var E* : Event* .
eq T |= X /\ Y = T |= X and T |= Y .
eq T |= X ++ Y = T |= X xor T |= Y .
eq (X /\ Y){E*} = X{E*} /\ Y{E*} .
eq (X ++ Y){E*} = X{E*} ++ Y{E*} .
eq [X /\ Y] = [X] and [Y] .
eq [X ++ Y] = [X] xor [Y] .
endfm

```

The statement “`extending LOGICS-BASIC`” imports the module `LOGICS-BASIC` with the reserve that its initial semantics can be extended. The operators “`and`” and “`xor`” come from the Maude’s built-in module `BOOL` which is automatically imported by any other module.

Operators are declared with special attributes, such as `assoc` and `comm`, which enable Maude to use its specialized efficient internal rewriting algorithms. Once the module above is loaded³ in Maude, reductions can be done as follows:

```

reduce a -> b /\ c <-> (a -> b) /\ (a -> c) . ***> should be true
reduce a <-> ! b . ***> should be a ++ b

```

Notice that one should first declare the constants `a`, `b` and `c`. The last six equations in the module `PROP-CALC` are related to the semantics of propositional calculus. The default evaluation strategy for `[_]` is eager, so `[X]` will first evaluate `X` using propositional calculus reasoning and then will apply one of the last two equations if needed; these equations will not be applied normally in practical reductions, they are useful only in the correctness proof stated by Theorem 16.

end from J.ASE’05

³Either by typing it or using the command “`in <filename>`”.

Chapter 3

Safety Properties

safety property = every violation occurs after a finite execution.

later, when talking about LTL, discuss the classification of safety properties in [124]

In the literature, what we call “prefixes” are also called “good prefixes”, while the rest of the prefixes are called “bad prefixes”.

Intuitively, a safety property of a system is one stating that the system cannot “go wrong”, or, as Lamport [125] put it, that the “bad thing” never happens. In other words, in order for a system to violate a safety property, it should eventually “go wrong” or the “bad thing” should eventually happen. There is a very strong relationship between safety properties and runtime monitoring: if a safety property is violated by a running system, then the violation should happen *during* the execution of the system, in a finite amount of time, so a monitor for that property observing the running system should be able to detect the violation; an additional point in the favor of monitoring is that, if a system violates a safety property at some moment during its execution, then there is no way for the system to continue its execution to eventually satisfy the property, so a monitor needs not wait for a better future once it detects a bad present/past.

State properties or assertions that need only the current state of the running system to check whether they are violated or not, such as “no division by 0”, or “ x positive”, or no deadlock, are common safety properties;

once violated, one can stop the computation or take corrective measures. However, there are also interesting safety properties that involve more than one state of the system, such as “if one uses resource x then one must have authenticated at some moment in the past”, or “any start of a process must be followed by a stop within 10 units of time”, or “take command from user only if the user has logged in at some moment in the past and has not logged out since then”, etc. Needless to say that the atomic events, or states, which form execution traces on which safety properties are defined, can be quite abstract: not all the details of a system execution are relevant for the particular safety property of interest. In the context of monitoring, these relevant events or states can be extracted by means of appropriate instrumentation of the system. For example, runtime monitoring systems such as Tracematches [5] and MOP [37] use aspect-oriented technology to “hook” relevant observation points and appropriate event filters in a system.

It is customary to define safety properties as properties over *infinite traces*, to capture the intuition that they are defined for systems that can potentially run forever, such as reactive systems. A point in favor of infinite traces is that finite traces can be regarded as special cases of infinite traces, namely ones that “stutter” indefinitely in their last state (see, for example, Abadi and Lamport [2, 3]). Infinite traces are particularly desirable when one specifies safety properties using formalisms that have infinite-trace semantics, such as linear temporal logics or corresponding automata.

While “infinity” is a convenient abstraction that is relatively broadly-accepted nowadays in mathematics and in theoretical foundations of computer science, there is no evidence so far that a system can have an infinite-trace behavior (we have not seen any). A disclaimer is in place here: we do *not* advocate finite-traces as a foundation for safety properties; all we try to do is to argue that, just because they can be seen as a special case of infinite traces, finite traces are not entirely uninteresting. For example, a safety property associated to a one-time-access key issued to a client can be “activate, then use at most once, then close”. Using regular patterns over the alphabet of relevant events $\Sigma = \{\text{activate}, \text{use}, \text{close}\}$, this safety property can be expressed as “ $\text{activate} \cdot (\epsilon + \text{use}) \cdot \text{close}$ ”; any trace that is not a prefix of the language of this regular expression violates the property, including any other activation or use of the key after it was closed. While these finite-trace safety properties can easily be expressed as infinite-trace safety properties, we believe that that would be more artificial than simply accepting that in practice we deal with many finite-trace safety properties.

In this section we discuss various approaches to formalize safety properties and show that they are ultimately directly or indirectly equivalent. We categorize them into finite-trace safety properties, infinite-trace safety properties, and finite- and infinite-trace safety properties:

1. Section 3.1 defines safety properties over finite traces as prefix closed properties. A subset of finite-trace safety properties, that we call *persistent*, contain only traces that “have a future” within the property, that is, finite traces that can be continued into other finite traces that are also in the safety property. Persistent safety properties appear to be the right finite-trace variant that corresponds faithfully to the more conventional infinite-trace safety properties. Even though persistent safety properties form a proper subset of finite-trace safety properties and each finite-trace safety property has a largest persistent safety property included in it, we show that there is in fact a bijection between safety properties and persistent safety properties by showing them both to have the cardinal of the continuum c .
2. In Section 3.2, we consider two standard infinite-trace definitions of a safety property, one based on the intuition that violating behaviors must manifest so after a finite number of events and the other based on the intuition of a safety property as a closed set in an appropriate topology over infinite-traces. We show them both equivalent to persistent safety properties over finite traces, by constructing an explicit bijection (as opposed to using cardinality arguments and infer the existence of a bijection); consequently, infinite-trace safety properties also have the cardinal of the continuum c . Since closed sets of real numbers are in a bijective correspondence with the real numbers, we indirectly rediscover Alpern and Schneider’s result [7] stating that infinite-trace safety properties correspond to closed sets in infinite-trace topology.
3. Section 3.3 considers safety properties defined over both finite and infinite traces. We discuss two definitions of such safety properties encountered in the literature, and, using cardinality arguments, we show their equivalence with safety properties over only finite traces. In particular, safety properties over finite and infinite traces also have the cardinality of the continuum c . We also show that prefix-closeness is not a sufficient condition to characterize (not even bijectively) such safety properties, by showing that there are significantly more (2^c) prefix-closed properties over finite and infinite traces than safety properties.

Therefore, each of the classes of safety properties is in bijection with the real numbers. Since there are so many safety properties, we can also insightfully conclude that there is *no* enumerable mechanism to define all the safety properties, because $\aleph_0 \leq c$. Therefore, particular logical or syntactic recursive formalisms can only define *some* of the safety properties, but not all of them.

3.1 Finite Traces

One of the most common intuitions for a safety property is as a prefix-closed set of finite traces. This captures best the intuition that once something bad happened, there is no way to recover: if $w \notin P$ then there is no u such that $P(wu)$, which is equivalent to saying that if $P(wu)$ then $P(w)$, which is equivalent to saying that P is prefix closed. From a monitoring perspective, a prefix closed property can be regarded as one containing all the good (complete or partial) behaviors of the observed system: once a state is encountered that does not form a good behavior together with the previously observed states, then a violation can be reported.

Definition 1 Let $\text{prefixes} : \Sigma^* \rightarrow \mathcal{P}(\Sigma^*)$ be the prefix function returning for any finite trace all its prefixes, and let $\text{prefixes} : \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$ be its corresponding closure operator that takes sets of finite traces and closes them under prefixes.

Note that $\text{prefixes} : \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$ is indeed a closure operator (Exercise 2): it is extensive ($P \subseteq \text{prefixes}(P)$), monotone ($P \subseteq P'$ implies $\text{prefixes}(P) \subseteq \text{prefixes}(P')$), and idempotent ($\text{prefixes}(\text{prefixes}(P)) = \text{prefixes}(P)$).

Definition 2 Let Safety^* be the set of finite-trace prefix-closed properties, that is, the set $\{P \in \mathcal{P}(\Sigma^*) \mid P = \text{prefixes}(P)\}$. In other words, Safety^* is the set of fixed points of the prefix operator $\text{prefixes} : \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$.

The star superscript in Safety^* reflects that its traces are finite; in the next section we will define a set Safety^ω of infinite-trace safety properties. Since $\text{prefixes}(P) \in \text{Safety}^*$ for any $P \in \mathcal{P}(\Sigma^*)$, we can assume from here on that $\text{prefixes} : \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$ is actually a function $\mathcal{P}(\Sigma^*) \rightarrow \text{Safety}^*$.

Example 1 Consider the one-time-access key safety property discussed above, saying that a client can “activate, then use at most once, and then

close” the key. If $\Sigma = \{activate, use, close\}$, then this safety property can be expressed as the finite set of finite words

$$\{\epsilon, activate, activate\ close, activate\ use, activate\ use\ close\}$$

No other behavior is allowed. Now suppose that the safety policy is extended to allow multiple uses of the key once activated, but still no further events once it is closed. The extended safety property has now infinitely many finite-traces:

$$\{\epsilon\} \cup \{activate\} \cdot \{use^n \mid n \in \mathbb{N}\} \cdot \{\epsilon, close\}.$$

Note that this property is indeed prefix-closed. A monitor in charge of online checking this safety property would report a violation if the first event is not *activate*, or if it encounters any second *activate* event, or if it encounters any event after a *close* event is observed, including another *close* event.

It is interesting to note that this finite-trace safety property encompasses both finite and infinite aspects. For example, it does not preclude behaviors in which one sees an *activate* event and then an arbitrary number of *use* events; *use* events can persist indefinitely after an *activate* event without violating the property. On the other hand, once a *close* event is encountered, no other event can be further seen. We will shortly see that the safety property above properly includes the *persistent* safety property $\{\epsilon\} \cup \{activate\ use^n \mid n \in \mathbb{N}\}$, which corresponds to the infinite-trace safety property $\{activate\ use^\omega\}$. \square

While prefix closeness seems to be the right requirement for a safety property, one can argue that it is not sufficient. For example, in the context of reactive systems that supposedly run forever, one may think of a safety property as one containing safe finite traces, that is, ones for which the reactive system can always find a way to continue its execution safely. The definition of safety properties above includes, among other safety properties, the empty set of traces as well as all prefix-closed *finite* sets of finite traces; any reactive system will eventually violate such safety properties, so one can say that the definition of safety property above is too generous.

We next define *persistent safety properties* as ones that always allow a future; intuitively, an observed reactive system that is in a safe state can always (if persistent enough) find a way to continue its execution to a next safe state. This notion is reminiscent of “feasibility”, a semantic characterization of fairness in [11], and of “machine closeness” [2, 158], also used in the context of fairness.

Definition 3 Let $\text{PersistentSafety}^*$ be the set of finite-trace persistent safety properties, that is, safety properties $P \in \text{Safety}^*$ such that if $P(w)$ for some $w \in \Sigma^*$ then there is some $a \in \Sigma$ such that $P(wa)$.

If a persistent safety property is non-empty, then note that it must contain an infinite number of words. The persistency aspect of a finite-trace safety property can be regarded, in some sense, as a liveness argument. Indeed, assuming that it is a “good thing” for a trace to be indefinitely continued, then a persistent safety property is one in which the “good thing” always eventually happens. If one takes the liberty to regard “stuck” computations as unfair, then the persistency aspect above can also be regarded as a fairness argument.

Another way to think of persistent safety properties is as a means to refer to infinite behaviors by means of finite traces. This view is, in some sense, dual to the more common approach to regard finite behaviors as infinite behaviors that stutter infinitely in a “last” state (see, for example, Abadi and Lamport [2, 3] for a formalization of such last-state infinite stuttering).

Note that if Σ is a degenerate set of events containing only one element, that is, if $|\Sigma| = 1$, then $|\text{Safety}^*| = \aleph_0$ and $|\text{PersistentSafety}^*| = 2$; indeed, if $\Sigma = \{a\}$ then Safety^* contains precisely the finite properties $a^{\leq n} = \{a^i \mid 0 \leq i \leq n\}$ for each $n \in \mathbb{N}$ plus the infinite property $\{a^n \mid n \in \mathbb{N}\}$, so a total of $\aleph_0 + 1 = \aleph_0$ properties, while $\text{PersistentSafety}^*$ contains only two properties, namely \emptyset and $\{a^n \mid n \in \mathbb{N}\}$. The case when there is only one event or state in Σ is neither interesting nor practical. Therefore, from here on in this paper we take the liberty to assume that $|\Sigma| \geq 2$. Since in practice Σ contains states or events generated by a computer, for simplicity in stating some of the subsequent results, we also take the liberty to assume that $|\Sigma| \leq \aleph_0$; therefore, Σ can be any finite or recursively enumerable set, including \mathbb{N} , \mathbb{N}_∞ , \mathbb{Q} , etc., but cannot be \mathbb{R} or any set “larger” than \mathbb{R} . With these assumptions, it follows that $|\Sigma^*| = \aleph_0$ (finite words are recursively enumerable) and $|\Sigma^\omega| = c$ (infinite streams have the cardinality of the continuum).

Proposition 1 Safety^* and $\text{PersistentSafety}^*$ are closed under union; Safety^* is also closed under intersection.

Proof: The union and the intersection of prefix-closed properties is also prefix-closed. Also, the union of persistent prefix-closed properties is also persistent. \square

The intersection of persistent safety properties may not be persistent:

Example 2 Let Σ be the set $\{0, 1\}$. Let $P = \{1^m \mid m \in \mathbb{N}\}$ and $P' = \{\epsilon\} \cup \{10^m \mid m \in \mathbb{N}\}$ be two persistent safety properties, where ϵ is the empty word (the word containing no letters). Then $P \cap P'$ is the finite safety property $\{\epsilon, 1\}$, which is not persistent. If one thinks that this happened because $P \cap P'$ does not contain any proper (i.e., non-empty) persistent property, then one can take instead the persistent safety properties $P = \{0^n \mid n \in \mathbb{N}\} \cdot \{1^m \mid m \in \mathbb{N}\}$ and $P' = \{0^n \mid n \in \mathbb{N}\} \cdot (\{\epsilon\} \cup \{10^m \mid m \in \mathbb{N}\})$, whose intersection is the safety property $\{0^n \mid n \in \mathbb{N}\} \cup \{0^n 1 \mid n \in \mathbb{N}\}$. This safety property is not persistent because its words ending in 1 cannot persist, but it contains the proper persistent safety property $\{0^n \mid n \in \mathbb{N}\}$. \square

Therefore, we can associate to any safety property in Safety^* a largest persistent safety property in $\text{PersistentSafety}^*$, by simply taking the union of all persistent safety properties that are included in the original safety property (the empty property is one of them, the smallest):

Definition 4 For a safety property $P \in \text{Safety}^*$, let $P^\circ \in \text{PersistentSafety}^*$ be the largest persistent safety property with $P^\circ \subseteq P$.

The following example shows that one may need to eliminate infinitely many words from a safety property in order to obtain a persistent safety property:

Example 3 Let $\Sigma = \{0, 1\}$ and let P be the safety property $\{0^n \mid n \in \mathbb{N}\} \cup \{0^n 1 \mid n \in \mathbb{N}\}$. Then P° can contain no word ending with a 1 and can contain all the words of 0's. Therefore, $P^\circ = \{0^n \mid n \in \mathbb{N}\}$. \square

Finite safety properties obviously cannot contain any non-empty persistent safety property, that is, $P^\circ = \emptyset$ if P is finite. But what if P is infinite? Is it always the case that it contains a non-empty persistent safety property? Interestingly, it turns out that this is true if and only if Σ is finite:

Proposition 2 If Σ is finite and P is a safety property containing infinitely many words, then $P^\circ \neq \emptyset$.

Proof: For each letter $a \in \Sigma$, let us define the *derivative of P wrt a* , written $\delta_a(P)$, as the language $\{w \in \Sigma^* \mid aw \in P\}$. Since

$$P = \{\epsilon\} \cup \bigcup_{a \in \Sigma} \{a\} \cdot \delta_a(P)$$

since Σ is finite, and since P is infinite, it follows that there is some $a_1 \in \Sigma$ such that $\delta_{a_1}(P)$ is infinite; note that $a_1 \in P$ since P is prefix closed. Similarly, since $\delta_{a_1}(P)$ is infinite, there is some $a_2 \in \Sigma$ such that $\delta_{a_2}(\delta_{a_1}(P))$ is infinite and $a_1 a_2 \in P$. Iterating this reasoning, we can find some $a_n \in \Sigma$ for each $n \in \mathbb{N}$, such that $a_1 a_2 \dots a_n \in P$ and $\delta_{a_n}(\dots(\delta_{a_2}(\delta_{a_1}(P)))\dots)$ is infinite, that is, the set $\{w \in \Sigma^* \mid a_1 a_2 \dots a_n w \in P\}$ is infinite. It is now easy to see that the set $\{a_1 a_2 \dots a_n \mid n \in \mathbb{N}\} \subseteq P$ is persistent. Therefore, $P^\circ \neq \emptyset$. \square

The following example shows that Σ must indeed be finite in order for the result above to hold:

Example 4 Consider some infinite set of events or states Σ . Then we can label distinct elements in Σ with distinct labels in $\mathbb{N} \cup \{\infty\}$. We only need these elements from Σ ; therefore, without loss of generality, we can assume that $\Sigma = \mathbb{N} \cup \{\infty\}$. Let P be the safety property

$$\{\epsilon\} \cup \{\infty n(n-1) \dots (m+1)m \mid 0 \leq m \leq n+1\},$$

where ϵ is the empty word (the word containing no letters) and $n \dots (n+1)$ is also the empty word for any $n \in \mathbb{N}$. Then P° is the empty property. Indeed, note that any persistent safety property P' included in P cannot have traces ending in 0, because those cannot be continued into other traces in P ; since P' cannot contain traces ending in 0, it cannot contain traces ending in 1 either, because such traces can only be continued with a 0 letter into traces in P , but those traces have already been decided that cannot be part of P' ; inductively, one can show that P' can contain no words ending in letters that are natural numbers in \mathbb{N} . Since the only trace in P ending in ∞ is ∞ itself and since ∞ can only be continued with a natural number letter into a trace in P but such trace cannot belong to P' , we deduce that P' can contain no word with letters in Σ . In particular, P° must be empty. \square

Even though we know that the largest persistent safety property P° included into a safety property P always exists because **PersistentSafety**^{*} is closed under union, we would like to have a more constructive way to obtain it. A first and obvious thing to do is to eliminate from P all the “stuck” computations, that is, those which cannot be added any new state to obtain a trace that is also in P . This removal step does not destroy the prefix-closeness of P , but it may reveal new computations which are stuck. By iteratively eliminating all the computations that get stuck in a finite number of steps, one would expect to obtain a persistent safety property,

namely precisely P° . It turns out that this is indeed true only if Σ is finite. If that is the case, then the following can also be used as an alternative definition of P° :

Proposition 3 *Given safety property $P \in \mathbf{Safety}^*$, then let P^- be the property $\{w \in P \mid (\exists a \in \Sigma) wa \in P\}$. Also, let $\{P_i \mid i \in \mathbb{N}\}$ be properties defined as $P_0 = P$ and $P_{i+1} = P_i^-$ for all $i \geq 0$. Then $P^\circ = \bigcap_{i \geq 0} P_i$ whenever Σ is finite.*

Proof: It is easy to see that if P is prefix-closed then $P^- \subseteq P$ is also prefix-closed, so P^- is also a property in \mathbf{Safety}^* . Therefore, the properties P_i form a sequence $P = P_0 \supseteq P_1 \supseteq P_2 \supseteq \dots$ of increasingly smaller safety properties.

Let us first prove that $\bigcap_{i \geq 0} P_i$ is a persistent safety property. Assume by contradiction that for some $w \in \bigcap_{i \geq 0} P_i$ there is no $a \in \Sigma$ such that $wa \in \bigcap_{i \geq 0} P_i$. In other words, we can find for each $a \in \Sigma$ some $i_a \geq 0$ such that $wa \notin P_{i_a}$. Since Σ is finite, we can let i be the largest among the natural numbers $i_a \in \mathbb{N}$ for all $a \in \Sigma$. Since $P_i \subseteq P_{i_a}$ for all $a \in \Sigma$, it should be clear that there is no $a \in \Sigma$ such that $wa \in P_i$, which means that $w \notin P_{i+1}$. This contradicts the fact that $w \in \bigcap_{i \geq 0} P_i$. Therefore, $\bigcap_{i \geq 0} P_i \in \mathbf{PersistentSafety}^*$.

Let us now prove that $\bigcap_{i \geq 0} P_i$ is the largest persistent safety property included in P . Let P' be any persistent safety property included in P . We show by induction on i that $P' \subseteq P_i$ for all $i \in \mathbb{N}$. The base case, $P' \subseteq P_0$, is obvious. Suppose that $P' \subseteq P_i$ for some $i \in \mathbb{N}$ and let $w \in P'$. Since P' is persistent, there is some $a \in \Sigma$ such that $wa \in P' \subseteq P_i$, which means that $w \in P_{i+1}$. Since w was chosen arbitrarily, it follows that $P' \subseteq P_{i+1}$. Therefore, $P' \subseteq \bigcap_{i \geq 0} P_i$. \square

We next show that the finiteness of Σ was a necessary requirement in order for the result above to hold. In other words, we show that if Σ is allowed to be infinite then we can find a safety property $P \in \mathbf{Safety}^*$ over Σ such that $P^\circ \in \mathbf{PersistentSafety}^*$ and $\bigcap_{i \geq 0} P_i \in \mathbf{Safety}^*$ are distinct. Since we showed in the proof of Proposition 3 that any persistent safety property P' is included in $\bigcap_{i \geq 0} P_i$, it follows that $P^\circ \subseteq \bigcap_{i \geq 0} P_i$. Since P° is the largest persistent safety property included in P , one can easily show that $P^\circ = (\bigcap_{i \geq 0} P_i)^\circ$. Therefore, it suffices to find a safety property P such that $\bigcap_{i \geq 0} P_i$ is not persistent, which is what we do in the next example:

Example 5 Consider the safety property P over infinite $\Sigma = \mathbb{N} \cup \{\infty\}$ discussed in Example 4, namely $\{\epsilon\} \cup \{\infty n(n-1) \dots (m+1)m \mid 0 \leq m \leq n\}$.

$n + 1\}$. Then one can easily show by induction on $i \in \mathbb{N}$ that the properties P_i defined in Proposition 3 are the sets $\{\epsilon\} \cup \{\infty n(n-1) \dots (m+1)m \mid i \leq m \leq n+1\}$; in other words, each P_i excludes from P all the words whose last letters are smaller than i when regarded as natural numbers. Then the intersection $\bigcap_{i \geq 0} P_i$ contains no trace ending in a natural number; the only possibility left is then $\bigcap_{i \geq 0} P_i = \{\epsilon, \infty\}$, which is different from $P^\circ = \emptyset$ (see Example 4).

One may argue that $P^\circ \neq \bigcap_{i \geq 0} P_i$ above happened precisely because P° was empty. One can instead pick the safety property $Q = \{0^n \mid n \in \mathbb{N}\} \cdot P$. Then one can show following the same idea as in Example 4 that $Q^\circ = \{0^n \mid n \in \mathbb{N}\}$. Further, one can show that $Q_i = \{0^n \mid n \in \mathbb{N}\} \cdot P_i$, so $\bigcap_{i \geq 0} Q_i = \{0^n \mid n \in \mathbb{N}\} \cup \{0^n \infty \mid n \in \mathbb{N}\}$, which is different from Q° . \square

Persistency is reminiscent of “feasibility” introduced by Apt et al. [11] in the context of fairness, and of “machine closeness” introduced by Abadi and Lamport [2, 3] (see also Schneider [158]) in the context of refinement. Let us use the terminology “machine closeness”: a property L (typically a liveness or a fairness property) is *machine closed* for a property M (typically given as the language of some state machine) iff L does not prohibit any of the observable runtime behaviors of M , that is, iff $\text{prefixes}(M) = \text{prefixes}(M \cap L)$; for example, if M is the total property (i.e., every event is possible at any moment, i.e., $M = \Sigma^*$) and L is the property stating that “always eventually event a ”, then any prefix of M can be continued to obtain a property satisfying L . Persistency is related to machine closeness in that a safety property P is persistent if and only if P° is machine closed for P . In other words, there is nothing P can do in a finite amount of time that P° cannot do. However, there is a caveat here: since liveness and fairness are inherently infinite-trace notions, machine closeness (or feasibility) have been introduced in the context of infinite-traces. On the other hand, persistency makes sense only in the context of finite traces.

It is clear that $\text{PersistentSafety}^*$ is properly included in Safety^* . Yet, we next show that, surprisingly, there is a bijective correspondence between Safety^* and $\text{PersistentSafety}^*$, both having the cardinal of the continuum:

Theorem 2 $|\text{PersistentSafety}^*| = |\text{Safety}^*| = c$.

Proof: Since Σ^* is recursively enumerable and since $2^{\aleph_0} = c$, we can readily infer that $|\text{PersistentSafety}^*| \leq |\text{Safety}^*| \leq |\mathcal{P}(\Sigma^*)| = c$.

Let us now define an injective function φ from the open interval of real numbers $(0, 1)$ to $\text{PersistentSafety}^*$. Since $|\Sigma| \geq 2$, let us distinguish

two different elements in Σ and let us label them $\bar{0}$ and $\bar{1}$. For a real $r \in (0, 1)$, let $\varphi(r)$ be the set $\{\bar{\alpha} \mid \alpha \in \{0, 1\}^* \text{ and } 0.\alpha < r\}$, where $0.\alpha$ is the (rational) number in $(0, 1)$ whose decimals in binary representation are α , and where $\bar{\alpha}$ is the word in Σ^* corresponding to α . Note that the set $\varphi(r) \in \mathcal{P}(\Sigma^*)$ is prefix-closed for any $r \in (0, 1)$, and that if $w \in \varphi(r)$ then also $w\bar{0} \in \varphi(r)$ (the latter holds since, by real numbers conventions, $0.\alpha = 0.\alpha 0$), so $\varphi(r) \in \text{PersistentSafety}^*$. Since the set of rationals with finite number of decimals in binary representation is dense in \mathbb{R} (i.e., it intersects any open interval in \mathbb{R}) and in particular in the interval $(0, 1)$, it follows that the function $\varphi : (0, 1) \rightarrow \text{PersistentSafety}^*$ is injective: indeed, if $r_1 \neq r_2 \in (0, 1)$, say $r_1 < r_2$, then there is some $\alpha \in \{0, 1\}^*$ such that $r_1 < 0.\alpha < r_2$, so $\varphi(r_1) \neq \varphi(r_2)$. Since the interval $(0, 1)$ has the cardinal of the continuum c , the existence of the injective function φ implies that $c \leq |\text{PersistentSafety}^*|$. By the Cantor-Bernstein-Schroeder theorem it follows that $|\text{PersistentSafety}^*| = |\text{Safety}^*| = c$. \square

From safety: *The proof above could have been rearranged to avoid the need to use the set $\text{PersistentSafety}^*$. However, we prefer to keep it for two reasons:*

1. *For finite-traces, persistent safety properties appear to be more natural in the context of reactive systems than just prefix closed properties;*
 2. *Persistent safety properties play a technical bridge role in the next section to show that the infinite-trace safety properties also have the cardinal c .*
-

With regards to finite-traces, persistent safety properties appear to be more natural in the context of reactive systems than just prefix-closed properties. Also, persistent safety properties play a technical bridge role in the next section to show that the infinite-trace safety properties also have the cardinal c .

3.2 Infinite Traces

The finite-trace safety properties defined above, persistent or not, rely on the intuition of a correct prefix: a safety property is identified with the set of all its finite prefixes. In the case of a persistent safety property, each “informal”

infinite acceptable behavior is captured by its infinite set of finite prefixes. Even though persistent safety properties appear to capture well in a finite-trace setting the intuition of safety in the context of (infinite-trace) reactive systems, one could argue that it does not say anything about unacceptable infinite traces. Indeed, one may think that persistent safety properties do not capture the intuition that if an infinite trace is unacceptable then there must be some finite prefix of it which is already unacceptable. In this section we show that there is in fact a bijection between safety properties over infinite traces and persistent safety properties over finite traces as we defined them in the previous section.

We start by extending the `prefixes` function to infinite traces:

Definition 5 *Let $\text{prefixes}: \Sigma^\omega \rightarrow \mathcal{P}(\Sigma^*)$ be the function returning for any infinite trace u all its finite prefixes $\text{prefixes}(u)$, and let $\text{prefixes}: \mathcal{P}(\Sigma^\omega) \rightarrow \mathcal{P}(\Sigma^*)$ be its corresponding extension to sets of infinite traces.*

Note that $\text{prefixes}(S) \in \text{PersistentSafety}^*$ for any $S \in \mathcal{P}(\Sigma^\omega)$, so `prefixes` is in fact a function $\mathcal{P}(\Sigma^\omega) \rightarrow \text{PersistentSafety}^*$.

The definition of safety properties over infinite traces below appears to be the most used definition of a safety property in the literature; at our knowledge, it was formally introduced by Alpern and Schneider [7], but they credit the insights of their definition to Lamport [125].

Definition 6 *Let Safety^ω be the set of infinite-trace properties $Q \in \mathcal{P}(\Sigma^\omega)$ s.t.: if $u \notin Q$ then there is a finite trace $w \in \text{prefixes}(u)$ s.t. $wv \notin Q$ for any $v \in \Sigma^\omega$.*

In other words, if an infinite behavior violates the safety property then there is some finite-trace “violation threshold”; once the violation threshold is reached, there is no chance to recover.

The following proposition can serve as an alternative and more compact definition of Safety^ω :

Proposition 4 $\text{Safety}^\omega = \{Q \in \mathcal{P}(\Sigma^\omega) \mid u \in Q \text{ iff } \text{prefixes}(u) \subseteq \text{prefixes}(Q)\}.$

Proof: Since $u \in Q$ implies $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$, the only thing left to show is that $Q \in \text{Safety}^\omega$ iff “ $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$ implies $u \in Q$ ”; the latter is equivalent to “ $u \notin Q$ implies $\text{prefixes}(u) \not\subseteq \text{prefixes}(Q)$ ”, which is further equivalent to “ $u \notin Q$ implies there is some $w \in \text{prefixes}(u)$ s.t. $w \notin \text{prefixes}(Q)$ ”, which is indeed equivalent to $Q \in \text{Safety}^\omega$. \square

Another common intuition for safety properties over infinite traces is as *closed* sets in the topology corresponding to Σ^ω . Alpern and Schneider captured formally this intuition for the first time in [7]; then it was used as a convenient definition of safety by Abadi and Lamport [2, 3] among others:

Definition 7 *An infinite sequence $u^{(1)}, u^{(2)}, \dots$, of infinite traces in Σ^ω converges to $u \in \Sigma^\omega$, or u is a limit of $u^{(1)}, u^{(2)}, \dots$, written $u = \lim_i u^{(i)}$, iff for all $m \geq 0$ there is an $n \geq 0$ such that $u_1^{(i)} u_2^{(i)} \dots u_m^{(i)} = u_1 u_2 \dots u_m$ for all $i \geq n$. If $Q \in \mathcal{P}(\Sigma^\omega)$ then \overline{Q} , the closure of Q , is the set $\{\lim_i u^{(i)} \mid u^{(i)} \in Q \text{ for all } i \in \mathbb{N}\}$.*

It can be easily shown that the overline closure above is indeed a closure operator on Σ^ω , that is, it is extensive ($Q \subseteq \overline{Q}$), monotone ($Q \subseteq Q'$ implies $\overline{Q} \subseteq \overline{Q'}$), and idempotent ($\overline{\overline{Q}} = \overline{Q}$); see Exercise 6.

Definition 8 *Let $\text{Safety}_{\lim}^\omega$ be the set of properties $\{Q \in \mathcal{P}(\Sigma^\omega) \mid Q = \overline{Q}\}$.*

As expected, the two infinite-trace safety property definitions are equivalent; we have not found any formal proof in the literature, so for the sake of completeness we give a simple proof here:

Proposition 5 $\text{Safety}_{\lim}^\omega = \text{Safety}^\omega$.

Proof: All we need to prove is that for any $Q \in \mathcal{P}(\Sigma^\omega)$ and any $u \in \Sigma^\omega$, $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$ iff $u = \lim_i u^{(i)}$ for some infinite sequence of infinite traces $u^{(1)}, u^{(2)}, \dots$ in Q . If $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$ then one can find for each $i \geq 0$ some $u^{(i)} \in Q$ such that $u_1 u_2 \dots u_i = u_1^{(i)} u_2^{(i)} \dots u_i^{(i)}$, so for each $m \geq 0$ one can pick $n = m$ such that $u_1 u_2 \dots u_m = u_1^{(i)} u_2^{(i)} \dots u_m^{(i)}$ for all $i \geq n$, so $u = \lim_i u^{(i)}$. Conversely, if $u = \lim_i u^{(i)}$ for some infinite sequence of infinite traces $u^{(1)}, u^{(2)}, \dots$ in Q , then for any $m \geq 0$ there is some $n \geq 0$ such that $u_1 u_2 \dots u_m = u_1^{(n)} u_2^{(n)} \dots u_m^{(n)}$, that is, for any prefix of u there is some $u' \in Q$ having the same prefix, that is, $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$. \square

The next result establishes the relationship between infinite-trace safety properties and finite-trace persistent safety properties, by proposing a concrete bijective mapping relating the two (as opposed to using cardinality arguments to indirectly show only the existence of such a mapping). Therefore, there is also a bijective correspondence between safety properties over infinite traces and the real numbers:

Note there are 2^c properties over Σ^ω

Theorem 3 $|\text{Safety}^\omega| = |\text{PersistentSafety}^*| = c$.

Proof: We show that there is a bijective function between the two sets of safety properties. Recall that $\text{prefixes}(S) \in \text{PersistentSafety}^*$ for any $S \in \mathcal{P}(\Sigma^\omega)$, that is, that prefixes is a function $\mathcal{P}(\Sigma^\omega) \rightarrow \text{PersistentSafety}^*$. Let $\text{prefixes} : \text{Safety}^\omega \rightarrow \text{PersistentSafety}^*$ be the restriction of this prefix function to Safety^ω . Let us also define a function $\omega : \text{PersistentSafety}^* \rightarrow \text{Safety}^\omega$ as follows: $\omega(P) = \{u \in \Sigma^\omega \mid \text{prefixes}(u) \subseteq P\}$. This function is well-defined: if $u \notin \omega(P)$ then by the definition of $\omega(P)$ there is some $w \in \text{prefixes}(u)$ such that $w \notin P$; since $w \in \text{prefixes}(wv)$ for any $v \in \Sigma^\omega$, it follows that $wv \notin \omega(P)$ for any $v \in \Sigma^\omega$.

We next show that prefixes and ω are inverse to each other. Let us first show that $\text{prefixes}(\omega(P)) = P$ for any $P \in \text{PersistentSafety}^*$. The inclusion $\text{prefixes}(\omega(P)) \subseteq P$ follows by the definition of $\omega(P)$: $\text{prefixes}(u) \subseteq P$ for any $u \in \omega(P)$. The inclusion $P \subseteq \text{prefixes}(\omega(P))$ follows from the fact that P is a persistent safety property: for any $w \in P$ one can iteratively build an infinite sequence v_1, v_2, \dots , such that $wv_1, wv_1v_2, \dots \in P$, so $wv_1v_2\dots \in \omega(P)$. Let us now show that $\omega(\text{prefixes}(Q)) = Q$ for any $Q \in \text{Safety}^\omega$. The inclusion $Q \subseteq \omega(\text{prefixes}(Q))$ is immediate. For the other inclusion, let $u \in \omega(\text{prefixes}(Q))$, that is, $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$. Suppose by contradiction that $u \notin Q$. Then there is some $w \in \text{prefixes}(u)$ such that $wv \notin Q$ for any $v \in \Sigma^\omega$. Since $w \in \text{prefixes}(u)$ and $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$, it follows that $w \in \text{prefixes}(Q)$, that is, that there is some $u' \in Q$ such that $u' = wv$ for some $v \in \Sigma^\omega$. This contradicts the fact that $wv \notin Q$ for any $v \in \Sigma^\omega$. Consequently, $u \in Q$.

The second part follows by Theorem 2. \square

3.3 Finite and Infinite Traces

It is also common to define safety properties as properties over both finite and infinite traces, the intuition for the finite traces being that of unfinished computations. For example, Lamport [126] extends the notion of safety in Definition 6 to properties over both finite and infinite traces, while Schneider et al [159, 82] give an alternative definition of safety over finite and infinite traces. We define both approaches shortly and then show their equivalence and their bijective correspondence with real numbers. Before that, we argue that the mix of finite and infinite traces is less trivial than it may appear,

by showing that there are significantly more prefix closed properties than in the case when only finite traces were considered.

Definition 9 Let $\text{PrefixClosed}^{*,\omega}$ be the set of prefix-closed sets of finite and infinite traces: for $Q \subseteq \Sigma^* \cup \Sigma^\omega$, $Q \in \text{PrefixClosed}^{*,\omega}$ iff $\text{prefixes}(Q) \subseteq Q$. Also, let $\text{PersistentPrefixClosed}^{*,\omega}$ be the set of persistent prefix-closed sets of finite and infinite traces: for $Q \in \text{PrefixClosed}^{*,\omega}$, it is the case that $Q \in \text{PersistentPrefixClosed}^{*,\omega} \iff$ if $Q(w)$ for some $w \in \Sigma^*$ then that there is some $a \in \Sigma$ such that $Q(wa)$.

The next result says that there is a bijective correspondence between prefix-closed and persistent prefix-closed properties also in the case of finite and infinite traces, but that there are exponentially more such properties than in the case of just finite traces:

Proposition 6 $|\text{PersistentPrefixClosed}^{*,\omega}| = |\text{PrefixClosed}^{*,\omega}| = 2^c$.

Proof: We show $2^c \leq |\text{PersistentPrefixClosed}^{*,\omega}| \leq |\text{PrefixClosed}^{*,\omega}| \leq 2^c$, where the middle inequality is immediate. For $2^c \leq |\text{PersistentPrefixClosed}^{*,\omega}|$, let us define $\varphi: \mathcal{P}((0, 1)) \rightarrow \text{PersistentPrefixClosed}^{*,\omega}$ as

$$\varphi(R) = \bigcup_{0.\alpha \in R} \{\bar{\alpha}\} \cup \text{prefixes}(\bar{\alpha})$$

where we assume for any real number in the interval $(0, 1)$ its decimal binary representation $0.\alpha$ with $\alpha \in \{0, 1\}^\omega$ (if the number is rational then α may contain infinitely many ending 0's), and $\bar{\alpha}$ is the infinite trace in Σ^ω replacing each 0 and 1 in α by $\bar{0}$ and $\bar{1}$, respectively, where $\bar{0}$ and $\bar{1}$ are two arbitrary but fixed distinct elements in Σ (recall that $|\Sigma| \geq 2$). Note that $\varphi(R)$ is well-defined: it is clearly prefix-closed and it is also persistent because its finite traces are exactly prefixes of infinite traces, so they admit continuations in $\varphi(R)$. It is easy to see that φ is injective. Since $|(0, 1)| = c$, we conclude that $2^c \leq |\text{PersistentPrefixClosed}^{*,\omega}|$.

To show $|\text{PrefixClosed}^{*,\omega}| \leq 2^c$, note that any property in $\text{PrefixClosed}^{*,\omega}$ is a union of a subset in Σ^* and a subset in Σ^ω , so $|\text{PrefixClosed}^{*,\omega}| \leq 2^{|\Sigma^*|} \cdot 2^{|\Sigma^\omega|}$. Since $|\Sigma^*| = \aleph_0$, $|\Sigma^\omega| = c$, $2^{\aleph_0} = c$, and $c \cdot 2^c = 2^c$ (by absorption of transfinite cardinals), we get that $|\text{PrefixClosed}^{*,\omega}| \leq 2^c$. \square

The fact that properties in $\text{PersistentPrefixClosed}^{*,\omega}$ contain also infinite traces was crucial in showing the injectivity of φ in the proof above. A similar construction for the finite trace setting does *not* work. Indeed, if

one tries to define a function $\varphi: \mathcal{P}((0, 1)) \rightarrow \text{PersistentSafety}^*$ as $\varphi(R) = \bigcup_{0.\alpha \in R} \text{prefixes}(\bar{\alpha})$, then one can show it well-defined but cannot show it injective: e.g., $\varphi((0, 0.5)) = \varphi((0, 0.5])$.

Since safety properties over finite and infinite traces are governed by the same intuitions as safety properties over only finite or over only infinite traces, the result above tells us that prefix closeness is not a sufficient condition to properly capture the safety properties. Schneider [159] proposes an additional condition in the context of his EM (execution monitoring) framework, namely that if an infinite trace is not in the property, then there is a finite prefix of it which is not in the property either. It is easy to see that this additional condition is equivalent to saying that an infinite trace is in the property whenever all its finite prefixes are in the property, which allows us to compactly define safety properties over finite and infinite traces in the EM style as follows:

Definition 10 $\text{Safety}_{\text{EM}}^{*,\omega} = \{Q \subseteq \Sigma^* \cup \Sigma^\omega \mid u \in Q \text{ iff } \text{prefixes}(u) \subseteq Q\}$.

Note that $\text{Safety}_{\text{EM}}^{*,\omega} \subset \text{PrefixClosed}^{*,\omega}$. We will shortly show that $\text{Safety}_{\text{EM}}^{*,\omega}$ is in fact much smaller than $\text{PrefixClosed}^{*,\omega}$, by showing that $|\text{Safety}_{\text{EM}}^{*,\omega}| = c$.

The consecrated definition of a safety property in the context of both finite and infinite traces is perhaps the one proposed by Lamport in [126], which relaxes the one in Definition 6 by allowing u to range over both finite and infinite traces:

Definition 11 *Let $\text{Safety}^{*,\omega}$ be the set of finite- and infinite-trace properties $\{Q \subseteq \Sigma^* \cup \Sigma^\omega \mid u \notin Q \Rightarrow (\exists w \in \text{prefixes}(u)) (\forall v \in \Sigma^* \cup \Sigma^\omega) wv \notin Q\}$*

Schneider informally stated in [159] that the two definitions of safety above are equivalent. It is not hard to show it formally:

Proposition 7 $\text{Safety}_{\text{EM}}^{*,\omega} = \text{Safety}^{*,\omega}$.

Proof: First note that $\text{Safety}^{*,\omega} \subseteq \text{PrefixClosed}^{*,\omega}$: if $wu \in Q \in \text{Safety}^{*,\omega}$ and $w \notin Q$ then there is some $w' \in \text{prefixes}(w)$, say $w = w'w''$, such that $w'v \notin Q$ for any v , in particular $w'w''u \notin Q$, which contradicts $wu \in Q$.

$\text{Safety}^{*,\omega} \subseteq \text{Safety}_{\text{EM}}^{*,\omega}$: let $Q \in \text{Safety}^{*,\omega}$ and $u \in \Sigma^* \cup \Sigma^\omega$ s.t. $\text{prefixes}(u) \subseteq Q$; if $u \notin Q$ then there is some $w \in \text{prefixes}(u)$ s.t. $wv \notin Q$ for any v , in particular for v the empty word, that is, $w \notin Q$, which contradicts $\text{prefixes}(u) \subseteq Q$.

$\text{Safety}_{\text{EM}}^{*,\omega} \subseteq \text{Safety}^{*,\omega}$: let $u \notin Q \in \text{Safety}_{\text{EM}}^{*,\omega}$; then $\text{prefixes}(u) \not\subseteq Q$, that is, there is some $w \in \text{prefixes}(u)$ s.t. $w \notin Q$; since Q is prefix-closed, it follows that $wv \notin Q$ for any $v \in \Sigma^* \cup \Sigma^\omega$. \square

We next show that there is a bijective correspondence between the safety properties over finite or infinite traces above and the finite trace safety properties in Section 3.1:

Theorem 4 $|\text{Safety}^{*,\omega}| = |\text{Safety}_{\text{EM}}^{*,\omega}| = |\text{Safety}^*| = c$.

Proof: $\text{Safety}^* \subset \text{Safety}_{\text{EM}}^{*,\omega}$ since the properties in $\text{Safety}_{\text{EM}}^{*,\omega}$ are prefix-closed, so $|\text{Safety}^*| \leq |\text{Safety}_{\text{EM}}^{*,\omega}|$.

Since the functions $\text{prefixes}: \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$ and $\text{prefixes}: \mathcal{P}(\Sigma^\omega) \rightarrow \mathcal{P}(\Sigma^*)$ have actual co-domains Safety^* and $\text{PersistentSafety}^*$, respectively, they can be organized as a function $\text{prefixes}: \text{Safety}_{\text{EM}}^{*,\omega} \rightarrow \text{Safety}^*$. Let us show that this function is injective. Let us assume $Q \neq Q' \in \text{Safety}_{\text{EM}}^{*,\omega}$, say $u \in Q$ and $u \notin Q'$, s.t. $\text{prefixes}(Q) = \text{prefixes}(Q')$. Since $u \in Q \in \text{Safety}_{\text{EM}}^{*,\omega}$ it follows that $\text{prefixes}(u) \subseteq \text{prefixes}(Q) \subseteq Q$, which implies that $\text{prefixes}(u) \subseteq \text{prefixes}(Q') \subseteq Q'$; since $Q' \in \text{Safety}_{\text{EM}}^{*,\omega}$, it follows that $u \in Q'$, contradiction. Therefore, $\text{prefixes}: \text{Safety}_{\text{EM}}^{*,\omega} \rightarrow \text{Safety}^*$ is injective, which proves that $\text{Safety}_{\text{EM}}^{*,\omega} \leq \text{Safety}^*$.

The rest follows by Proposition 7 and Theorem 2. \square

3.4 “Always Past” Characterization

Another common way to specify safety properties is by giving an arbitrary property on finite traces, not necessarily prefix closed, and then to require that any acceptable behavior must have all its finite prefixes in the given property. A particularly frequent case is when one specifies the property of the finite-prefixes using the past-time fragment of linear temporal logics (LTL). For example, Manna and Pnueli [129] call the resulting “always (past LTL)” properties *safety formulae*; many other authors, including ourselves, adopted the terminology “safety formula” from Manna and Pnueli, although some qualify it as “LTL safety formula”. An example of an LTL safety formula is “always (b implies eventually in the past a)”, written using LTL notation as “ $\square(b \rightarrow \diamond a)$ ”; here the past time formula “ $b \rightarrow \diamond a$ ” compactly specifies all the finite-traces

$$\{ws w' s' \mid w, w' \in \Sigma^*, s, s' \in \Sigma, a(s) \text{ and } b(s') \text{ hold}\} \cup \{ws \mid w \in \Sigma^*, s \in \Sigma, b(s) \text{ does not hold}\}.$$

From safety: *We will investigate the case when safety properties are expressed as LTL safety formulae, as well as optimal monitoring techniques for such safety properties, in Section 10.7.*

In the remainder of this section we assume that the past time prefix properties are given as ordinary sets of finite-traces (so we make abstraction of how these properties are expressed) and show not only that the resulting “always past” properties are safety properties, but also that any safety properties can be expressed as an “always past” property. This holds for all the variants of safety properties (i.e., over finite traces, over infinite traces, or over both finite and infinite traces).

Definition 12 *Let $P \subseteq \Sigma^*$ be any property over finite traces. Then we define the “always past” property $\Box P$ as follows:*

- (finite traces) $\{w \in \Sigma^* \mid \text{prefixes}(w) \subseteq P\}$; and
- (infinite traces) $\{u \in \Sigma^\omega \mid \text{prefixes}(u) \subseteq P\}$; and
- (finite and infinite traces) $\{u \in \Sigma^* \cup \Sigma^\omega \mid \text{prefixes}(u) \subseteq P\}$.

Let Safety_\Box^ , $\text{Safety}_\Box^\omega$ and $\text{Safety}_\Box^{*,\omega}$ be the corresponding sets of properties.*

From safety: *In Section 10.7 we show that the language $\mathcal{L}(\Box\varphi)$, that corresponds to the LTL safety formula $\Box\varphi$ for φ some past-time LTL formula, is a property in $\text{Safety}_\Box^\omega$. If one was interested in a finite-trace or in a both finite and infinite trace semantics of LTL, then one could have shown that $\mathcal{L}(\Box\varphi) \in \text{Safety}_\Box^*$ or that $\mathcal{L}(\Box\varphi) \in \text{Safety}_\Box^{*,\omega}$.*

Intuitively, one can regard the square “ \Box ” as a closure operator. Technically, it is not precisely a closure operator because it does not operate on the same set: it takes finite-trace properties to any of the three types of properties considered. Since prefixes takes properties back to finite-trace properties, we can show the following result saying that the square is a “closure operator via prefixes ”, and that safety properties are precisely the sets of words which are closed this way:

Proposition 8 *The following hold for all three types of safety properties:*

- $\Box(\text{prefixes}(\Box P)) = \Box P$ for any $P \subseteq \Sigma^*$;
- Q is a safety property iff $\Box(\text{prefixes}(Q)) = Q$.

Proof: Left as an exercise to the reader. See Exercise 5. \square

We next show that the “always past” properties are all safety properties and, moreover, that any safety property can be expressed as an “always past” property:

Theorem 5 *The following hold:*

- $\text{Safety}_{\square}^* = \text{Safety}^*$,
- $\text{Safety}_{\square}^{\omega} = \text{Safety}^{\omega}$, and
- $\text{Safety}_{\square}^{*,\omega} = \text{Safety}^{*,\omega}$.

Therefore, each of the “always past” safety properties have the cardinal c .

Proof: We prove each of the equalities by double inclusion.

$\text{Safety}_{\square}^* \subseteq \text{Safety}^*$. It is true because any property $\square P$ in $\text{Safety}_{\square}^*$ is prefix-closed.

$\text{Safety}^* \subseteq \text{Safety}_{\square}^*$. If $P \in \text{Safety}^*$ then we claim that $P = \square P$, so $P \in \text{Safety}_{\square}^*$. Indeed, since P is prefix-closed, $\text{prefixes}(w) \subseteq P$ for any $w \in P$, so $w \in \square P$; also, since $w \in \text{prefixes}(w)$, it follows that for any $w \in \square P$, $w \in P$.

$\text{Safety}_{\square}^{\omega} \subseteq \text{Safety}^{\omega}$. Let $\square P$ be an “always past” property in $\text{Safety}_{\square}^{\omega}$, and let u be an infinite trace in Σ^{ω} such that $u \notin \square P$. Then it follows that $\text{prefixes}(u) \not\subseteq P$, that is, there is some $w \in \text{prefixes}(u)$ such that $w \notin P$. Since $w \in \text{prefixes}(wv)$ for any $v \in \Sigma^{\omega}$, it means that there is no $v \in \square P$ such that $\text{prefixes}(wv) \subseteq P$, that is, there is no $v \in \Sigma^{\omega}$ such that $wv \in \square P$. Therefore, $\square P \in \text{Safety}^{\omega}$.

$\text{Safety}^{\omega} \subseteq \text{Safety}_{\square}^{\omega}$. If $Q \in \text{Safety}^{\omega}$ then we claim that $Q = \square \text{prefixes}(Q)$. The inclusion $Q \subseteq \square \text{prefixes}(Q)$ is clear, because $u \in Q$ implies $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$. For the other inclusion, note that if $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$ for some $u \in \Sigma^{\omega}$, then u must be in Q : if $u \notin Q$ then by the definition of $Q \in \text{Safety}^{\omega}$, there is some $w \in \text{prefixes}(u)$ which cannot be completed into an infinite trace in Q , which contradicts $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$.

$\text{Safety}_{\square}^{*,\omega} \subseteq \text{Safety}^{*,\omega}$. By Proposition 7, it suffices to show that $\text{Safety}_{\square}^{*,\omega} \subseteq \text{Safety}_{\text{EM}}^{*,\omega}$. Let $\square P$ be an “always past” property in $\text{Safety}_{\square}^{*,\omega}$, and let $u \in \Sigma^* \cup \Sigma^{\omega}$ such that $\text{prefixes}(u) \subseteq \text{prefixes}(\square P)$. Since $\text{prefixes}(\square P) \subseteq P$, it follows that $u \in \square P$; therefore, $\square P \in \text{Safety}_{\text{EM}}^{*,\omega}$.

$\text{Safety}^{*,\omega} \subseteq \text{Safety}_{\square}^{*,\omega}$. It is straightforward to see that $Q \in \text{Safety}_{\text{EM}}^{*,\omega}$ implies $Q = \square \text{prefixes}(Q)$.

The cardinality part follows by Theorems 2, 3, and 4. \square

Proposition 8 and Theorem 5 give yet another characterization for safety properties over any of the three combinations of traces, namely one in the style of the equivalent formulation of safety over infinite traces in Proposition 4: Q is a safety property iff it contains precisely the words whose prefixes are in $\text{prefixes}(Q)$.

Exercises

Exercise 2 The $\text{prefixes}: \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$ in Definition 1 is a closure operator: it is extensive ($P \subseteq \text{prefixes}(P)$), monotone ($P \subseteq P'$ implies $\text{prefixes}(P) \subseteq \text{prefixes}(P')$), and idempotent ($\text{prefixes}(\text{prefixes}(P)) = \text{prefixes}(P)$).

Exercise 3 (Counter-)Example 4 showed that the finiteness of Σ was necessary in order for Proposition 2 to hold, by defining a property P over $\Sigma = \mathbb{N} \cup \{\infty\}$ in which all non-empty words start with ∞ . Can we remove ∞ from Σ and from all the words in P ? Why, or why not?

Exercise 4 Same like Exercise 3, but for Example 5 instead of Example 4.

Exercise 5 Prove Proposition 8.

Exercise 6 The “closure under limits” operation in Definition 7 is indeed a closure operator on Σ^ω : it is extensive ($Q \subseteq \overline{Q}$), monotone ($Q \subseteq Q'$ implies $\overline{Q} \subseteq \overline{Q'}$), and idempotent ($\overline{\overline{Q}} = \overline{Q}$).

Chapter 4

Monitoring

In this section we give yet another characterization of safety properties, namely as monitorable properties. Specifically, we formally define a monitor as a (possibly infinite) state machine without final states but with a partial transition function, and then we show that safety properties are precisely the properties that can be monitored with such monitors. We then elaborate on the problem of defining the complexity of monitoring a safety property, discussing some pitfalls and guiding principles, and show that monitoring a safety property can be an arbitrarily hard problem. Finally, we give a more compact and mathematical equivalent definition of a monitor, which may be useful in further foundational efforts in this area.

Relate our definition of a monitor with Schneider's security automata

4.1 Specifying Safety Properties as Monitors

Safety properties are difficult to work with as flat sets of finite or infinite words, not only because they can contain infinitely many words, but also because such a flat representation is inconvenient for further analysis. It is important therefore to *specify* safety properties using formalisms that are easier to represent and reason about.

From safety: *The next sections in this paper investigate several dedicated formalisms that proved to be convenient in specifying safety, such as finite state machines, regular expressions and temporal logics, together with corresponding limitations and efficient monitor synthesis techniques.*

Formalisms known to be useful for specifying safety properties include regular expressions and temporal logics, which can be efficiently translated into finite-state machines which can then be used as monitors. In this section we formalize the intuitive notion of a *monitor* as a special state machine and give yet another characterization of safety properties, namely as *monitorable properties*. Since monitorable properties are completely defined by their monitors, it follows that *all* safety properties can be specified by their corresponding monitors.

Recall that we work under the assumption that Σ is a set of events or program states such that $|\Sigma| \leq \aleph_0$.

Definition 13 *A Σ -monitor, or just a monitor (when Σ is understood), is a triple $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$, where S is a set of states, $s_0 \in S$ is the initial state, and M is a deterministic partial transition function.*

Therefore, a monitor as defined above is nothing but a deterministic state machine without final states. Moreover, the set of states is allowed to be infinite, and the transition function has no complexity requirements (it can even be undecidable). We could have defined monitors to be standard state machines, but the subsequent technical developments would have been slightly more involved.

From safety: *In fact, we aim at shortly giving an even more compact definition of a monitor, that we will call canonical monitor, which appears to be sufficient to capture any safety property.*

The intuition for a monitor is the expected one: the monitor is driven by events generated by the observed program (the letters in Σ)—each newly received event drives the monitor from its current state to some other state, as indicated by the transition function M ; if the monitor ever gets stuck, that is, if the transition function M is undefined on the current state and the current event, then the monitored property is declared violated at that point by the monitor.

For any partial function $M : S \times \Sigma \rightarrow S$, we obey the following common notational convention. If $s \in S$ and $w = w_1 w_2 \dots w_k \in \Sigma^*$, we write “ $M(s, w) \downarrow$ ” whenever $M(s, w)$ is defined, that is, whenever $M(s, w_1)$ and $M(M(s, w_1), w_2)$ and ... and $M(\dots(M(s, w_1), w_2) \dots, w_k)$ are all defined, which is nothing but only saying that $M(\dots(M(s, w_1), w_2) \dots, w_k)$ is defined. If we write $M(s, w) = s'$ for some $s' \in S$, then, as expected, we mean that $M(\dots(M(s, w_1), w_2) \dots, w_k)$ is defined and equal to s' .

A monitor specifies a finite-trace property, an infinite-trace property, as well as a finite- and infinite-trace property:

Definition 14 *Given a monitor $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$, we define the following properties:*

- $\mathcal{L}^*(\mathcal{M}) = \{w \in \Sigma^* \mid M(s_0, w) \downarrow\}$,
- $\mathcal{L}^\omega(\mathcal{M}) = \{u \in \Sigma^\omega \mid M(s_0, w) \downarrow \text{ for all } w \in \text{prefixes}(u)\}$, and
- $\mathcal{L}^{*,\omega}(\mathcal{M}) = \mathcal{L}^*(\mathcal{M}) \cup \mathcal{L}^\omega(\mathcal{M})$.

We call $\mathcal{L}^*(\mathcal{M})$ the finite-trace property specified by \mathcal{M} , call $\mathcal{L}^\omega(\mathcal{M})$ the infinite-trace property specified by \mathcal{M} , and call $\mathcal{L}^{*,\omega}(\mathcal{M})$ the finite- and infinite-trace property specified by \mathcal{M} . Also, we let

$$\mathcal{S}_{\mathcal{M}} = \{s \in S \mid (\exists w \in \Sigma^*) M(s_0, w) = s\}$$

denote the set of reachable states of \mathcal{M} .

A *monitorable* property is a property which can be specified by a monitor. We next capture this intuitive notion formally:

Definition 15 *For a property $P \subseteq \Sigma^* \cup \Sigma^\omega$, we let $\text{Monitors}(P)$ be the set of monitors $\{\mathcal{M} \mid \mathcal{L}^{*,\omega}(\mathcal{M}) = P\}$. If $\text{Monitors}(P) \neq \emptyset$ then P is called monitorable and the elements of $\text{Monitors}(P)$ are called monitors of P . We define the following classes of properties:*

- $\text{Monitorable}^* = \{P \subseteq \Sigma^* \mid P \text{ monitorable}\}$,
- $\text{Monitorable}^\omega = \{P \subseteq \Sigma^\omega \mid P \text{ monitorable}\}$, and
- $\text{Monitorable}^{*,\omega} = \{P \subseteq \Sigma^* \cup \Sigma^\omega \mid P \text{ monitorable}\}$.

The notion of persistence can also be adapted to monitors:

Definition 16 A monitor $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$ is persistent iff for any reachable state $s \in \mathcal{S}_{\mathcal{M}}$, there is an $a \in \Sigma$ such that $M(s, a) \downarrow$. Let

- $\text{PersistentMonitorable}^* = \{\mathcal{L}^*(\mathcal{M}) \mid \mathcal{M} \text{ persistent}\}$

be the set of finite-trace properties monitorable by persistent monitors.

Our next goal is to show that each monitor admits a largest persistent “submonitor”. To formalize it, we lift the conventional partial order relation on partial functions to monitors:

Definition 17 If $\mathcal{M}_1 = (S, s_0, M_1 : S \times \Sigma \rightarrow S)$ and $\mathcal{M}_2 = (S, s_0, M_2 : S \times \Sigma \rightarrow S)$ are two monitors sharing the same states and initial state, then let $\mathcal{M}_1 \sqsubseteq \mathcal{M}_2$, read \mathcal{M}_1 a submonitor of \mathcal{M}_2 , iff for any $s \in S$ and any $a \in \Sigma$, if $M_1(s, a)$ is defined then $M_2(s, a)$ is also defined and $M_2(s, a) = M_1(s, a)$.

The above can be easily generalized to allow \mathcal{M}_1 to only have a subset of the states of \mathcal{M}_2 , but we found that generalization unnecessary so far.

The above partial-order on monitors allows us to use conventional mathematics to obtain the largest persistent sub-monitor of a monitor:

Proposition 9 $(\{\mathcal{K} \mid \mathcal{K} \sqsubseteq \mathcal{M} \text{ and } \mathcal{K} \text{ persistent}\}, \sqsubseteq)$ is a complete (join) semilattice for any monitor \mathcal{M} .

Proof: If $\{\mathcal{K}_i = (S, s_0, K_i : S \times \Sigma \rightarrow S) \in \mathcal{M}\}_{i \in I}$ is a set of persistent monitors, then their supremum (or join) is the monitor $\mathcal{K} = (S, s_0, K : S \times \Sigma \rightarrow S)$ where $K(s, a) = s'$ iff there is some $i \in I$ such that $K_i(s, a) = s'$. It is easy to see that \mathcal{K} is a well-defined monitor and that it is persistent. \square

Since complete semilattices have maximum elements, the following definition is fully justified:

Definition 18 For any monitor $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$, we let $\mathcal{M}^\circ = (S, s_0, M^\circ : S \times \Sigma \rightarrow S)$ be the \sqsubseteq -maximal element of the complete lattice $(\{\mathcal{K} \mid \mathcal{K} \sqsubseteq \mathcal{M} \text{ and } \mathcal{K} \text{ persistent}\}, \sqsubseteq)$.

We next show that, as expected, there is a tight relationship between persistent safety properties (Definition 3) and persistent canonical monitors.

Proposition 10 Let $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$. Then the following hold:

- $\mathcal{L}^\omega(\mathcal{M}) = \mathcal{L}^\omega(\mathcal{M}^\circ)$,

- $\mathcal{L}^*(\mathcal{M}^\circ) = \mathcal{L}^*(\mathcal{M})^\circ$, and
- \mathcal{M} persistent iff $\mathcal{L}^*(\mathcal{M})$ persistent.

Proof: The first property can be shown by the following sequence of equivalences: $u \in \mathcal{L}^\omega(\mathcal{M})$ iff $M(s_0, w) \downarrow$ for all $w \in \text{prefixes}(u)$, iff there is some persistent monitor $\mathcal{K} \subseteq \mathcal{M}$ such as $K(s_0, w) \downarrow$ for all $w \in \text{prefixes}(u)$, iff $M^\circ(s_0, w) \downarrow$ for all $w \in \text{prefixes}(u)$, iff $u \in \mathcal{L}^\omega(\mathcal{M}^\circ)$.

The second property can be shown as follows: $w \in \mathcal{L}^*(\mathcal{M}^\circ)$ iff $M^\circ(s_0, w) \downarrow$, iff there is some $u \in \mathcal{L}^\omega(\mathcal{M}^\circ)$ such that $w \in \text{prefixes}(u)$ (because \mathcal{M}° is persistent), iff there is some $u \in \mathcal{L}^\omega(\mathcal{M})$ such that $w \in \text{prefixes}(u)$ (by the first property), iff there is some $u \in \Sigma^\omega$ such that $w \in \text{prefixes}(u) \subseteq \mathcal{L}^*(\mathcal{M})$, iff there is some $u \in \Sigma^\omega$ such that $w \in \text{prefixes}(u) \subseteq \mathcal{L}^*(\mathcal{M})^\circ$ (because $\text{prefixes}(u)$ is a persistent safety property), iff $w \in \mathcal{L}^*(\mathcal{M})^\circ$.

Finally, the third property is an immediate consequence of the second, noticing that \mathcal{M} is persistent iff it is equal to \mathcal{M}° , and that $\mathcal{L}^*(\mathcal{M})$ is persistent iff it is equal to $\mathcal{L}^*(\mathcal{M})^\circ$. \square

Theorem 6 *The following hold:*

- $\text{Monitorable}^* = \text{Safety}^*$,
- $\text{Monitorable}^\omega = \text{Safety}^\omega$,
- $\text{Monitorable}^{*,\omega} = \text{Safety}^{*,\omega}$, and
- $\text{PersistentMonitorable}^* = \text{PersistentSafety}^*$.

Proof: First, note that the following hold for any monitor \mathcal{M} :

- $\mathcal{L}^*(\mathcal{M}) \in \text{Safety}^*$,
- $\mathcal{L}^\omega(\mathcal{M}) \in \text{Safety}^\omega$, and
- $\mathcal{L}^{*,\omega}(\mathcal{M}) \in \text{Safety}^{*,\omega}$.

These all follow by Theorem 5: taking P in Definition 12 to be the property $\{w \in \Sigma^* \mid M(s_0, w) \downarrow\}$, then $\Box P$ over finite traces is precisely $\mathcal{L}^*(\mathcal{M})$, over infinite traces is precisely $\mathcal{L}^\omega(\mathcal{M})$, and over finite and infinite traces is precisely $\mathcal{L}^{*,\omega}(\mathcal{M})$, so the three languages are in Safety_\Box^* , $\text{Safety}_\Box^\omega$, and $\text{Safety}_\Box^{*,\omega}$, respectively. Therefore, $\text{Monitorable}^* \subseteq \text{Safety}^*$, $\text{Monitorable}^\omega \subseteq \text{Safety}^\omega$, and $\text{Monitorable}^{*,\omega} \subseteq \text{Safety}^{*,\omega}$.

Second, note that we can associate a default monitor \mathcal{M}_P to any finite-trace property $P \subseteq \Sigma^*$, namely $(S_P, \epsilon, M_P : S_P \times \Sigma \rightarrow S_P)$, where $S_P = \text{prefixes}(P)$, ϵ is the empty word, and $M_P(w, a)$ is defined iff $wa \in \text{prefixes}(P)$, and in that case $M_P(w, a) = wa$. Moreover, it is easy to check that

- $\mathcal{L}^*(\mathcal{M}_P) = \{w \in \Sigma^* \mid \text{prefixes}(w) \subseteq P\} = \Box P$ (over finite traces) ,
- $\mathcal{L}^\omega(\mathcal{M}_P) = \{u \in \Sigma^\omega \mid \text{prefixes}(u) \subseteq P\} = \Box P$ (over infinite traces),
- $\mathcal{L}^{*,\omega}(\mathcal{M}_P) = \{u \in \Sigma^* \cup \Sigma^\omega \mid \text{prefixes}(u) \subseteq P\} = \Box P$ (over both finite and infinite traces).

Since P was chosen arbitrarily, it follows then by Theorem 5 that $\text{Safety}^* \subseteq \text{Monitorable}^*$, $\text{Safety}^\omega \subseteq \text{Monitorable}^\omega$, and $\text{Safety}^{*,\omega} \subseteq \text{Monitorable}^{*,\omega}$.

Finally, the equality $\text{PersistentMonitorable}^* = \text{PersistentSafety}^*$ follows by the first fact and by Proposition 10. \square

4.2 Complexity of Monitoring a Safety Property

We here address the problem of defining the complexity of monitoring. Before we give our definition, let us first discuss some pitfalls in defining this notion. Our definition for the complexity of monitoring resulted as a consequence of trying to avoid these pitfalls. Let P be a safety property.

Pitfall 1.

The complexity of monitoring P is nothing but the complexity of checking, for an input word $w \in \Sigma^$, whether $w \in \text{prefixes}(P)$.*

This would be an easy to formulate decision problem, but, unfortunately, does not capture well the intuition of monitoring, because it does not require that the word w be processed incrementally, as its letters become available from the observed system. Incremental processing of letters can make a huge difference in both how complex monitoring is and how monitoring complexity can be defined. For example, it is well-known that the membership problem of a finite word to the language of an extended regular expression (ERE), i.e., a regular expression extended with complement operators, is a polynomial problem (the classic algorithm by Hopcroft and Ullman [104] runs in space $O(m^2 \cdot n)$ and time $O(m^3 \cdot n)$, where m is the size of the word and n that of the expression). However,

From safety: as shown in Section 7

there are EREs defining safety properties whose monitoring requires non-elementary space and time. Of course, this non-elementary lower-bound is expressed only as a function of the size of the ERE representing the safety property; it does not take into account the size of the monitored trace. This leads us to our first guiding principle:

Principle 1.

The complexity of monitoring a safety property P should depend only upon P , not upon the trace being monitored.

Indeed, since monitoring is a process that involves potentially unbounded traces, if the complexity of monitoring a property P were expressed as a function of the execution trace as well, then that complexity measure would be close to meaningless in practice, because monitoring reactive systems would have unbounded complexity. For example, consider an operating system monitoring some safety property on how its resources are being used by the various running processes; what one would like to know here is what is the runtime overhead of monitoring that safety property at each relevant event, and not the obvious fact that the more the operating system runs the larger the total runtime overhead is.

Nevertheless, one can admittedly argue that it would still be useful to know how complex the monitoring of P against a given finite trace w is, in terms of both the size of (some representation of) P and the size of w ; however, this is nothing but a conventional membership test decision problem, that has nothing to do with monitoring. If one picks some arbitrary off-the-shelf efficient algorithm for membership testing and uses that at each newly received event on the existing finite execution trace, then one may obtain a “monitoring” algorithm whose complexity to process each event grows in time, as events are processed. In the context of monitoring a reactive system, that means that eventually the monitoring process may become unfeasible, regardless of how many resources are initially available and regardless of how efficient the membership testing algorithm is. What one needs in order for the monitoring process to stay feasible regardless of how many events are observed, is a special membership algorithm that processes each event as received and whose state or processing time does not increase potentially unbounded as events are received. Therefore, one needs an algorithm which, if it takes resources R to check w , then it takes at most $R + \Delta$ to check a one-event continuation wa of w , where Δ *does not depend on w* . In other words, one needs a *monitor for P of complexity Δ* .

Pitfall 2.

P is typically infinite, so the complexity of monitoring P should be a function of the size of some finite specification, or representation, of P.

Indeed, since Principle 1 tells us that the complexity of monitoring P is a function of P only and not of the monitored trace, one may be tempted to conclude that it is a function of the *size* of some convenient encoding of P . There are at least two problems with this approach, that we discuss below.

- One problem is that the same property P can be specified in many different ways as a structure of finite size; for example, it can be specified as a regular expression, as an extended regular expression, as a temporal logic formula, as an ordinary automaton, as a push-down automaton, etc. These formalisms may represent P as specifications of quite different sizes. Which is the most appropriate? It is, nevertheless, interesting and important to study the complexity of monitoring safety properties expressed using different specification formalisms, as a function of the property representation size, because that can give us an idea of the amount of resources needed to monitor a particular specification. However, one should be aware that such a complexity measure is an attribute of the corresponding specification formalisms, not of the specified property itself. Indeed, the higher this complexity measure for a particular formalism, the higher the encoding strength of safety properties in that formalism: for example, the complexity of monitoring safety properties expressed as EREs is non-elementary in the size of the original ERE, while the complexity of monitoring the same property expressed as an ordinary regular expression is linear in the size of the regular expression. Does that mean that one can monitor safety properties expressed as regular expressions non-elementarily more efficiently than one can monitor safety properties expressed as EREs? Of course not, because EREs and regular expressions have the same expressiveness, so they specify exactly the same safety properties. All it means is that EREs can express safety-properties non-elementarily more compactly than ordinary regular expressions.
- Another problem with this approach is that apparently appropriate representations of P may be significantly larger than it takes to monitor P . One may say, for example, that, whenever possible, a natural way to specify a particular safety property is as a finite-state machine, e.g.,

as a monitor like in Definition 13 . To be more concrete, consider that the safety property P_n saying “every 2^n -th event is a ” is specified as a monitor of 2^n states that transits with any event from each state to the next one, except for the 2^n -th state, which has only one transition, with event a , back to state 1. Therefore, the size of this representation of P_n is $\Omega(2^n)$. Assuming that each state takes n bits of storage (for example, assume that states are exactly the binary encodings of the numbers 1, 2, 3, ..., 2^n) and that the next state can be calculated from the current state in linear complexity with the size of the state (which is true in our scenario), then it is clear that the actual complexity of monitoring P_n is $O(n)$. If the complexity of monitoring P_n were a function of the size of the specification of P_n , then one could wrongly conclude that the complexity of monitoring “every 2^n -th event is a ” is $O(2^n)$.

Therefore, a safety property P has an inherent complexity w.r.t. monitoring, complexity which has nothing to do with how P is represented, or encoded, or specified. It is that inherent complexity attribute of safety properties that we are after here. From the discussion above, we draw our second guiding principle:

Principle 2.

The monitoring complexity of a safety property P is an attribute of P alone, not a function of the size of some adhoc representation of P .

By Theorem 6, safety properties are precisely those properties that are monitorable, that is, those properties P for which there are (finite-state or not) monitors $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$ whose (finite-trace, infinite-trace, or finite- and infinite-trace—this depends upon the type of P) language is precisely P . Any algorithm, program or system that one may come up with to be used as a monitor for P , can be organized as a monitor of the form $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$ for P . Consequently, the complexity of monitoring P cannot be smaller than the functional complexity of the partial function $M : S \times \Sigma \rightarrow S$ corresponding to some “best” monitor \mathcal{M} for P ; if there are no additional restrictions, then by “best” monitor we mean the one whose functional complexity of M is smallest. In particular, if there is no monitor for P whose transition partial function M is decidable, then we can safely say that the problem of monitoring P is undecidable. This discussion leads to the following:

Pitfall 3.

The complexity of monitoring P is the functional complexity of function M , where $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$ is the “best” monitor for P .

Since safety properties are precisely the monitorable properties, this appears to be a very natural definition for the complexity of monitoring. While the functional complexity of the monitor function is indeed important because it directly influences the efficiency of monitoring, it is *not* a sufficient measure for the complexity of monitoring. That is because the functional complexity of M only says how complex M is in terms of the size of *its input*; it does not say anything about how large the state of the monitor can grow in time. For example, the rewriting-based monitoring algorithm for EREs from [147], whose states are EREs and whose transition is a derivative operation of functional complexity $O(n^2)$ taking an ERE of size n into an ERE of size $O(n^2)$. It would be very misleading to say that the complexity of monitoring EREs is $O(n^2)$, because it may sound much better than it actually is: the n^2 factor accumulates as events are processed. Any monitor for EREs, including the one based on derivatives, eventually requires non-elementary resources (in the size of the ERE) to process a new event.

Therefore, while the complexity of the function M being executed at each newly received event by a monitor \mathcal{M} is definitely a necessary and important factor to be considered when defining the complexity of monitoring using \mathcal{M} , it is not sufficient. One also needs to take into account the size of the input that is being passed to the monitoring function, that is, the size of the monitor state together with the size of the received event. In particular, a monitor storing all the observed trace has unbounded complexity, say ∞ , even though its monitoring function has trivial complexity (e.g., the “event storing” function has linear complexity). More generally, if a property admits no finite-state monitor, then we’d like to say that its monitoring complexity is ∞ : indeed, for any monitor for such a property and for any amount of resources R , there is some sequence of events that would lead the monitor to a state that needs more than R resources to be stored or computed. These observations lead us to the following:

Principle 3. The complexity of monitoring P is a function of both the functional complexity of M and of the size of the states in S , where $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$ is an appropriately chosen (“best”) monitor for P .

We next follow the three principles above and derive our definition for the complexity of monitoring a safety property P . Before that, let us first define the complexity of monitoring a safety property using a particular monitor for that property, or in other words, let us first define the complexity of a monitor.

During a monitoring session using a monitor, at any moment in time one needs to store at least one state, namely the state that the monitor is currently in. When receiving a new event, the monitor launches its transition function on the current state and the received input. Therefore, the (worst-case) complexity of monitoring with $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$ could be defined as

$$\max\{FC(M(s, a)) \mid s \in S, a \in \Sigma\},$$

where $FC(M(s, a))$ is the functional complexity of evaluating M on state s and event a , as a function of the sizes of s and a . In other words, the worst-case monitoring complexity of a particular monitor is the maximal functional complexity that its transition function has on any state and any input; this functional complexity is expressed as a function of the size of the pair (state,event). In order for such a definition to make sense formally, one would need to define or axiomatize the size of monitor states and the size of events. Since in order to distinguish N elements one needs $\log(N)$ space, we deduce that one needs at least $\log(|S|)$ space to store the state of the monitor in its worst-case monitoring scenario (each state in S is reachable).

Definition 19 *Given a monitor $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$, we define the complexity of monitoring \mathcal{M} , written $C_{Mon}(\mathcal{M})$, as the function*

$$FC(M)(\log |S|) : \mathbb{N} \rightarrow \mathbb{N},$$

which is the “uncurried” version applied on $\log |S|$ of the worst-case functional complexity $FC(M) : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ of the partial function M as a function of the size of the pair (state,event) being passed to it.

We assume that the complexity of monitoring a safety property P is the worst-case complexity of monitoring it using some appropriate, “best” monitor for P :

$$\min\{\max\{FC(M(s, a)) \mid s \in S, a \in \Sigma\} \mid \mathcal{M} = (S, s_0, M) \in \text{Monitors}(P)\},$$

From safety: where $FC(M(s, a))$ is the functional complexity of evaluating M on state s and event a , as a function of the sizes of s and a . In other words, we assume that the complexity of monitoring a safety property P is the worst-case complexity of monitoring it using some appropriate, “best” monitor for P . The worst-case monitoring complexity of a particular monitor is the maximal functional complexity that its transition function has on any state and any input; this functional complexity is expressed as a function of the size of the pair (state, event). Therefore, in order for such a definition to make sense formally, one would need to define or axiomatize the size of monitor states and the size of events.

This gives us the following:

Definition 20 We let

$$C_{Mon}(P) = \min\{FC(M) \circ \langle \log(|S|), 1_\Sigma \rangle \mid \mathcal{M} = (S, s_0, M) \in \text{Monitors}(P)\}$$

be the complexity of monitoring a safety property P .

4.3 Monitoring Safety Properties is Arbitrarily Hard

We show that the problem of monitoring a safety property can be arbitrarily complex. The previous section tells us that there are as many safety properties as real numbers. Therefore, it is not surprising that some of them can be very hard or impossible to monitor. In this section we formalize this intuitive argument. Our approach is to show that we can associate a safety property P_S to any set of natural numbers S , such that monitoring that safety property is as hard as checking membership of arbitrary natural numbers to S . The result then follows from the fact that checking memberships of natural numbers to sets of natural numbers is a problem that can be arbitrarily complex.

Theorem 2 indirectly says that we can associate a persistent safety property to any set of natural numbers (sets of natural numbers are in a bijective correspondence with the real numbers). However, it is not clear how that safety property looks and neither how to monitor it. We next give a more concrete mapping from sets of natural numbers to (persistent) safety properties and show that monitoring the property is equivalent to

testing membership to the set. It suffices to assume that Σ contains only two elements, say $\Sigma = \{0, 1\}$.

Definition 21 Let $P_- : \mathcal{P}(\mathbb{N}) \rightarrow \text{PersistentSafety}^*$ be the mapping defined as follows: for any $S \subseteq \mathbb{N}$, let P_S be the set $1^* \cup \{1^k 0 \mid k \in S\} \cdot \{0, 1\}^*$.

It is easy to see that P_S is a persistent safety property over finite traces. Also, it is easy to see that the bijection in the proof of Theorem 3 associates to P_S the safety property over infinite traces $1^\omega \cup \{1^k 0 \mid k \in S\} \cdot \{0, 1\}^\omega$.

Let us now investigate the problem of monitoring P_S .

Proposition 11 For any $S \subseteq \mathbb{N}$, monitoring P_S is equivalent to deciding membership of natural numbers to S .

Proof: If M_S is an oracle deciding membership of natural numbers to S , that is, if $M_S(n)$ is true iff $n \in S$, then one can build a monitor for P_S as follows: for a given trace, incrementally read and count the number of prefix 1's; if no 0 is ever observed then monitor indefinitely without reporting any violation; when a first 0 is observed, if any, ask if $M(k)$, where k is the number of 1's observed; if $M(k)$ is false, then report violation; if $M(k)$ is true, then continue monitoring indefinitely and never report violation. It is clear that this is indeed a monitor for P_S .

Conversely, if we had any monitor for P_S then we could build a decision procedure for membership to S as follows: given $k \in \mathbb{N}$, send to the monitor a sequence of k ones followed by a 0; if the monitor reports violation then deduce that $k \notin S$; if the monitor does not report violation, then deduce that $k \in S$. It is clear that this is a decision procedure for membership to S .

The proof works for both persistent safety properties over finite traces and for safety properties over infinite traces. \square

The claim in the title of this section follows now from the fact that the set S of natural numbers can be chosen so that its membership problem is arbitrarily complex. For example, since there are as many subsets of natural numbers as real numbers while there are only as many Turing machines as natural numbers, it follows that there are many (exponentially) more sets of natural numbers that are not recognized by Turing machines than those that are. In particular, there are sets of natural numbers corresponding to any degree in the arithmetic hierarchy, i.e., to predicates $A(k)$ of the form $(Q_1 k_1)(Q_2 k_2) \cdots (Q_n k_n) R(k, k_1, k_2, \dots, k_n)$, where Q_1, Q_2, \dots, Q_n are alternating (universal or existential) quantifiers and R is a recursive/decidable

relation: for A such a predicate, let S_A be the set of natural numbers $\{k \mid A(k)\}$. Recall that if Q_1 is \forall then A is called a Π_n property, while if Q_1 is \exists then A is called a Σ_n property. In particular, $\Sigma_0 = \Pi_0$ and they contain precisely the recursive/decidable properties, Σ_1 contains precisely the class of recursively enumerable problems, Π_1 contains precisely the co-recursively enumerable problems, etc.; a standard Π_2 problem is TOTALITY: given $k \in \mathbb{N}$, is it true that Turing machine with Gödel number k terminates on all inputs? Since each level in the arithmetic hierarchy contains problems strictly harder than problems on the previous layer (because $\Sigma_n \cup \Pi_n \subsetneq \Sigma_{n+1} \cap \Pi_{n+1}$), the arithmetic hierarchy gives us a universe of safety properties whose monitoring can be arbitrarily hard.

Within the decidable fragment, as expected, monitoring safety properties can also have any complexity. Indeed, pick for example any NP-complete problem and let S be the set of inputs (coded as natural numbers) for which the problem has a positive answer; then, as explained in the proof of Proposition 11, monitoring P_S against input 1^k0 is equivalent to deciding membership of k to S , which is further equivalent to answering the NP-complete problem on input k . Of course, in practice a particular (implementation of a) monitor can be more complex than the corresponding membership problem; for example, monitors corresponding to NP-complete problems are most likely exponential. Also, note that a monitor for P_S needs not necessarily do its complex computation on an input 1^k0 when it encounters the 0. It can perform intermediate computations as it reads the prefix 1's and thus pay a lesser computational price when the 0 is encountered. What Proposition 11 says is that the *total* complexity to process the input 1^k0 can be no lower than the complexity of checking whether $k \in S$.

4.4 Canonical Monitors

We conclude this section with an alternative definition of a monitor, called *canonical monitor*, which is more compact than our previous definition and which appears to be sufficient to capture any safety property. We do not make any use of this alternative definition in this paper, but it may serve as a basis for further foundational endeavors in this area.

The set of states S of a monitor $(S, s_0, M : S \times \Sigma \rightarrow S)$ are typically enumerable, so they can be very well replaced with natural numbers. Moreover, the initial state s_0 can be encoded, by convention, as the first natural number, 0. A monitor then becomes nothing but a partial function $\mathbb{N} \times \Sigma \rightarrow \mathbb{N}$. We

therefore rightfully call these particular monitors *canonical*:

Definition 22 A canonical Σ -monitor is a partial function $\mathcal{N} : \mathbb{N} \times \Sigma \rightarrow \mathbb{N}$. Let $\mathcal{S}_{\mathcal{N}} = \{n \mid (\exists w) \mathcal{N}(0, w) = n\}$ be the states of \mathcal{N} . As before, let

- $\mathcal{L}^*(\mathcal{N}) = \{w \in \Sigma^* \mid \mathcal{N}(0, w) \downarrow\}$,
- $\mathcal{L}^\omega(\mathcal{N}) = \{u \in \Sigma^\omega \mid \mathcal{N}(0, w) \downarrow \text{ for all } w \in \text{prefixes}(u)\}$, and
- $\mathcal{L}^{*,\omega}(\mathcal{N}) = \mathcal{L}^*(\mathcal{N}) \cup \mathcal{L}^\omega(\mathcal{N})$.

Although the set of states S in a monitor $(S, s_0, M : S \times \Sigma \rightarrow S)$ is allowed to have any cardinal while the states in canonical monitors are restricted to natural numbers, it turns out that canonical monitors can in fact express all monitorable properties:

Proposition 12 A property $P \subseteq \Sigma^*$ (resp. $P \subseteq \Sigma^\omega$, resp. $P \subseteq \Sigma^* \cup \Sigma^\omega$) is monitorable iff there is some canonical monitor \mathcal{N} such that $P = \mathcal{L}^*(\mathcal{N})$ (resp. $P = \mathcal{L}^\omega(\mathcal{N})$, resp. $P = \mathcal{L}^{*,\omega}(\mathcal{N})$).

Proof: Since any canonical monitor is a monitor, it follows that any property specifiable by a canonical monitor is indeed monitorable. For the converse, let P be a property monitorable by some monitor $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$. Since $|\Sigma| \leq \aleph_0$, we can enumerate all the states of \mathcal{M} that can be reached from s_0 with its transition function M . There are many different ways to do this (e.g., in breadth-first order, in depth-first order, etc.), but these are all ultimately irrelevant. If we let $S^r = \{s_0, s_1, s_2, \dots\}$ denote the resulting set of reachable states, then it is easy to first note that the monitor $\mathcal{M}^r = (S^r, s_0, M : S^r \times \Sigma \rightarrow S^r)$ specifies the same property P as \mathcal{M} , and second note that \mathcal{M}^r specifies the same property as the canonical monitor $\mathcal{N} : \mathbb{N} \times \Sigma \rightarrow \mathbb{N}$ defined by $\mathcal{N}(i, a) = j$ iff $M(s_i, a) = s_j$. \square

Exercises

Exercise 7 Define a canonical monitor for the property

“A file can only be accessed if it is open.”

That is, the file can only be accessed if it was opened at some moment in the past and it was not closed since then. Suppose Σ consists of the events/actions $\{o, a, c\}$, where o stands for “file open”, a for “file access”, and c for “file close”.

Chapter 5

Event/Trace Observation

Chapter 6

Monitor Synthesis: Finite State Machines (FSM)

Currently, this chapter has a lot of duplication. The material has been collected from 2 papers, but not rationally merged yet.

discuss also DFA, NFA, using NFA as monitor, REs, RE2NFA, derivatives

6.1 Binary Transition Trees (BTT)

6.1.1 Multi-Transition and Binary Transition Tree Finite State Machines

To keep the runtime overhead of monitors low, it is crucial to do as little computation as possible in order to proceed to the next state. In the sequel we assume that finite state monitors are desired to efficiently change their state (when a new event is received) to one of possible states s_1, s_2, \dots, s_n , under the knowledge that a transition to each such state is enabled deterministically by some Boolean formula, p_1, p_2, \dots, p_n , respectively, on atomic state predicates.

Definition 23 *Let S be a set whose elements are called states, and let A be another set, whose elements are called atomic predicates. Let $\{s_1, s_2, \dots, s_n\} \subseteq S$ and let p_1, p_2, \dots, p_n be propositions over atoms in A (using the usual*

Boolean combinators presented in Subsection 2.1.2), with the property that exactly one of them is true at any moment, that is, $p_1 \vee p_2 \vee \dots \vee p_n$ holds and for any distinct p_i, p_j , it is the case that $p_i \rightarrow \neg p_j$ holds. Then we call the n -tuple $[p_1?s_1, p_2?s_2, \dots, p_n?s_n]$ a multi-transition (MT) over states S and atomic predicates (or simply atoms) A . Let $MT(S, E)$ be the set of multi-transitions over states S and atoms A .

Since the rest of the paper is concerned with rather theoretical results, from now on we use mathematical symbols for the Boolean operators instead of ASCII symbols like in Subsection 2.1.2. The intuition underlying multi-transitions is straightforward: depending on which of the propositions p_1, p_2, \dots, p_n is true at a given moment, a corresponding transition to exactly one of the states s_1, s_2, \dots, s_n will take place. Formally,

Definition 24 Maps $\theta : A \rightarrow \{\text{true}, \text{false}\}$ are called events from now on. With the notation above, given an event θ , we define a map $\theta_{MT} : MT(S, A) \rightarrow S$ as $\theta_{MT}([p_1?s_1, p_2?s_2, \dots, p_n?s_n]) = s_i$, where $\theta(p_i) = \text{true}$; notice that $\theta(p_j) = \text{false}$ for any $1 \leq j \neq i \leq n$.

Definition 25 Given an event $\theta : A \rightarrow \{\text{true}, \text{false}\}$, let e_θ denote the list of atomic predicates a with $\theta(a) = \text{true}$. There is an obvious correspondence between events as maps $A \rightarrow \{\text{true}, \text{false}\}$ and events as lists of atomic predicates, which justifies our implementation of events in Subsection 2.1.2. We take the liberty to use either the map or the list notation for events from now on in the paper. We let \mathcal{E} denote the set of all events and call lists, or words, $e_{\theta_1} \dots e_{\theta_n} \in \mathcal{E}^*$ (finite) traces; this is also consistent with our mechanical Maude ASCII notation in Subsection 2.1.2, except that we prefer not to separate events by commas in traces from now on, to avoid mathematical notational conflicts.

We next define binary transition trees.

Definition 26 Under the same notations as in the previous definition, a binary transition tree (BTT) over states S and atoms A is a term over syntax

$$BTT ::= S \mid (A ? BTT : BTT).$$

We let $BTT(S, A)$ denote the set of binary transition trees over states S and atoms A . Given an event $\theta : A \rightarrow \{\text{true}, \text{false}\}$, we define a map

$\theta_{BTT} : BTT(S, A) \rightarrow S$ inductively as follows:

$$\begin{aligned} \theta_{BTT}(s) &= s \text{ for any } s \in S, \\ \theta_{BTT}(a ? b_1 : b_2) &= \theta_{BTT}(b_1) \text{ if } \theta(a) = \text{true, and} \\ \theta_{BTT}(a ? b_1 : b_2) &= \theta_{BTT}(b_2) \text{ if } \theta(a) = \text{false.} \end{aligned}$$

A binary transition tree b in $BTT(S, A)$ is said to implement a multi-transition t in $MT(S, A)$ if and only if $\theta_{BTT}(b) = \theta_{MT}(t)$ for any map $\theta : A \rightarrow \{\text{true}, \text{false}\}$.

BTTs generalize BDDs [29], which can be obtained by taking $S = \{true, false\}$. As an example of BTT, $a_1 ? a_2 ? s_1 : a_3 ? false : s_2 : a_3 ? s_2 : true$ says “eval a_1 ; if a_1 then (eval a_2 ; if a_2 then go to state s_1 else (eval a_3 ; if a_3 then go to state $false$ else go to state s_2)) else (eval a_3 ; if a_3 then go to state s_2 else go to state $true$)”. Note that $true$ and $false$ are just some special states. Depending on the application they can have different meanings, but in our applications $true$ typically means that the monitoring requirement has been fulfilled, while $false$ means that it has been violated. It is often convenient to represent BTTs graphically, such as the one in Figure 6.1.

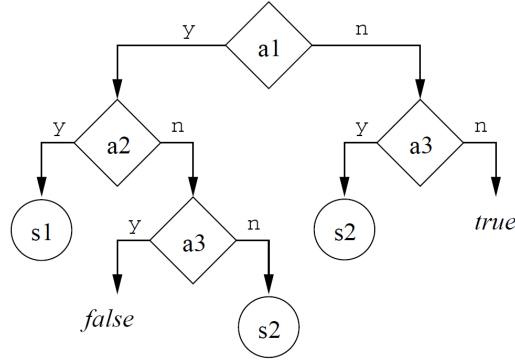


Figure 6.1: Graphical representation for the BTT $a1 ? a2 ? s1 : a3 ? false : s2 : a3 ? s2 : true$.

Definition 27 A multi-transition finite state machine (*MT-FSM*) is a triple (S, A, μ) , where S is a set of states, A is a set of atomic predicates, and μ is a map from $S - \{\text{true}, \text{false}\}$ to $MT(S, A)$. If S contains *true* and/or *false*, then $\mu(\text{true}) = [\text{true?true}]$ and/or $\mu(\text{false}) = [\text{true?false}]$, respectively. A binary

transition tree finite state machine (*BTT-FSM*) is a triple (S, A, β) , where S and A are like before and β is a map from $S - \{\text{true}, \text{false}\}$ to $BTT(S, A)$. If S contains *true* and/or *false*, then it is the case that $\beta(\text{true}) = [\text{true}]$ and/or $\beta(\text{false}) = [\text{false}]$, respectively. For an event $\theta : A \rightarrow \{\text{true}, \text{false}\}$ in any of these machines, we let $s \xrightarrow{\theta} s'$ denote the fact that $\theta_{MT}(\mu(s)) = s'$ or $\theta_{BTT}(\beta(s)) = s'$, respectively. We take the liberty to call $s \xrightarrow{\theta} s'$ a transition. Also, we may write $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \dots s_n \xrightarrow{\theta_n} s_{n+1}$ and call it a sequence of transitions whenever $s_1 \xrightarrow{\theta_1} s_2, \dots, s_n \xrightarrow{\theta_n} s_{n+1}$ are transitions.

Note that S is not required to contain the special states *true* and *false*, but if it contains them, these states transit only to themselves. The finite state machine notions above are intended to abstract the intuitive concept of a monitor. The states in S are monitor states, and some of them can trigger side effects in practice, such as messages sent or reported to users, actions taken, feedback sent to the monitored program for guidance purposes, etc.

Definition 28 If $\text{true} \in S$ then a sequence of transitions $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \dots s_n \xrightarrow{\theta_n} \text{true}$ is called a *validating sequence* (for s_1); if $\text{false} \in S$ then $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \dots s_n \xrightarrow{\theta_n} \text{false}$ is called an *invalidating sequence* (for s_1). These notions naturally extend to corresponding finite traces of events $e_{\theta_1} \dots e_{\theta_n}$.

BTT-FSMs can and should be thought of as efficient “implementations” of MT-FSMs, in the sense that they implement multi-transitions as binary transition trees. These allow one to reduce the amount of computation in a monitor implementing a BTT-FSM to only evaluate at most all the atomic predicates in a given state in order to decide to which state to go next; however, it will only very rarely be the case that *all* the atomic predicates need to be evaluated.

A natural question is why one should bother at all then with defining and investigating MT-FSMs. Indeed, what one really looks for in the context of FSM monitoring is efficient BTT-FSMs. However, MT-FSMs are nevertheless an important intermediate concept as it will become clearer later in the paper. This is because they have the nice property of *state mergeability*, which allows one to elegantly generate MT-FSMs from logical formulae. By state mergeability we mean the following. Suppose that during a monitor generation process, such as that for LTL that will be presented in Subsection 8.4.1, one proves that states s and s' are “logically equivalent” (for now, equivalence can be interpreted intuitively: they have the same behavior),

and that s and s' have the multi-transitions $[p_1?s_1, p_2?s_2, \dots, p_n?s_n]$ and $[p'_1?s'_1, p'_2?s'_2, \dots, p'_n?s'_{n'}]$. Then we can merge s and s' into one state whose multi-transition is $Merge([p_1?s_1, p_2?s_2, \dots, p_n?s_n], [p'_1?s'_1, p'_2?s'_2, \dots, p'_n?s'_{n'}])$, defined as follows:

$$\begin{aligned}
& Merge([p_1?s_1, p_2?s_2, \dots, p_n?s_n], [p'_1?s'_1, p'_2?s'_2, \dots, p'_n?s'_{n'}]) \\
& \text{contains all choices } p?s'', \text{ where } s'' \text{ is a state in } \{s_1, s_2, \dots, s_n\} \cup \\
& \{s'_1, s'_2, \dots, s'_{n'}\} \text{ and} \\
& \begin{aligned}
& \bullet p \text{ is } p_i \text{ when } s'' = s_i \text{ for some } 1 \leq i \leq n \text{ and } s'' \neq s'_{i'} \text{ for all} \\
& \quad 1 \leq i' \leq n', \text{ or} \\
& \bullet p \text{ is } p'_{i'} \text{ when } s'' = s'_{i'} \text{ for some } 1 \leq i' \leq n' \text{ and } s'' \neq s_i \text{ for} \\
& \quad \text{all } 1 \leq i \leq n, \text{ or} \\
& \bullet p \text{ is } p_i \vee p'_{i'} \text{ when } s'' = s_i \text{ for some } 1 \leq i \leq n \text{ and } s'' = s'_{i'} \\
& \quad \text{for some } 1 \leq i' \leq n'.
\end{aligned}
\end{aligned}$$

It is easy to see that this multi-transition merging operation is well defined, that is,

Proposition 13 *$Merge([p_1?s_1, p_2?s_2, \dots, p_n?s_n], [p'_1?s'_1, p'_2?s'_2, \dots, p'_n?s'_{n'}])$ is a well-formed multi-transition whenever both $[p_1?s_1, p_2?s_2, \dots, p_n?s_n]$ and $[p'_1?s'_1, p'_2?s'_2, \dots, p'_n?s'_{n'}]$ are well-formed multi-transitions. Therefore, MERGE can be seen as a function $\mathcal{P}_f(MT(S, A)) \rightarrow MT(S, A)$, where \mathcal{P}_f is the finite powerset operator.*

There are situations when a definite answer, *true* or *false* is desired at the end of the monitoring session, as it will be in the case of LTL. As explained in Section 8.1, the intuition for the last event in an execution trace is that the trace is infinite and stationary in that last event. This seems to be the best and simplest assumption about future when a monitoring session is ended.

Discussion about variants of finite-trace LTL needed.

For such situations, we enrich our definitions of MT-FSM and BTT-FSM with support for terminal events:

Definition 29 *A terminating multi-transition finite state machine (abbreviated MT-FSM*) is a tuple (S, A, μ, μ^*) , where (S, A, μ) is an MT-FSM and*

μ^* is a map from $S - \{true, false\}$ to $MT(\{true, false\}, A)$. A terminating binary transition tree finite state machine (*BTT-FSM**) is a tuple (S, A, β, β^*) , where (S, A, β) is a BTT-FSM and β^* is a map from $S - \{true, false\}$ to $BTT(\{true, false\}, A)$. For a given event $\theta : A \rightarrow \{true, false\}$ in any of these finite state machines, we let $s \xrightarrow{\theta^*} true$ (or $false$) denote the fact that $\theta_{MT}(\mu^*(s)) = true$ (or $false$) or $\theta_{BTT}(\beta^*(s)) = true$ (or $false$), respectively. We call $s \xrightarrow{\theta^*} true$ (or $false$) a terminating transition. A sequence $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \dots s_n \xrightarrow{\theta_n^*} true$ is called an accepting sequence (for s_1) and a sequence $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \dots s_n \xrightarrow{\theta_n^*} false$ is called a rejecting sequence (for s_1). These notions also naturally extend to corresponding finite traces of events $e_{\theta_1} \dots e_{\theta_n}$.

Languages can be associated to states in MT-FSM*s or BTT-FSM*s as finite words of events.

Definition 30 Given a state $s \in S$ in an MT-FSM* or in a BTT-FSM* M , we let $\mathcal{L}_M(s)$ denote the set of finite traces $e_{\theta_1} \dots e_{\theta_n}$ with the property that $s \xrightarrow{\theta_1} \dots \xrightarrow{\theta_n^*} true$ is an accepting sequence in M . If a state $s_0 \in S$ is desired to be initial, then we write it at the end of the tuple, such as (S, A, μ, μ^*, s_0) or $(S, A, \beta, \beta^*, s_0)$, and let \mathcal{L}_M denote $\mathcal{L}_M(s_0)$. If $s_0 \xrightarrow{\theta_1} s_1 \dots \xrightarrow{\theta_n} s_n$ is a sequence of transitions from the initial state, then $e_{\theta_1} \dots e_{\theta_n}$ is called a valid prefix if and only if $e_{\theta_1} \dots e_{\theta_n} t \in \mathcal{L}_M$ for any (empty or not) trace t , and it is called an invalid prefix if and only if $e_{\theta_1} \dots e_{\theta_n} t \notin \mathcal{L}_M$ for any trace t .

The following is immediate.

Proposition 14 If $true \in S$ then $\mathcal{L}_M(true) = \mathcal{E}^*$, and if $false \in S$ then $\mathcal{L}_M(false) = \emptyset$. If $s \xrightarrow{\theta} s'$ in M then $\mathcal{L}_M(s') = \{t \mid e_{\theta}t \in \mathcal{L}_M(s)\}$; more generally, if $s \xrightarrow{\theta_1} s_1 \dots \xrightarrow{\theta_n} s_n$ is a sequence of transitions in M then $\mathcal{L}_M(s_n) = \{t \mid e_{\theta_1} \dots e_{\theta_n} t \in \mathcal{L}_M(s)\}$. In particular, if $s = s_0$ then $e_{\theta_1} \dots e_{\theta_n}$ is a valid prefix if and only if $\mathcal{L}_M(s_n) = \mathcal{E}^*$, and it is an invalid prefix if and only if $\mathcal{L}_M(s_n) = \emptyset$.

6.1.2 From MT-FSMs to BTT-FSMs

Supposing that one has encoded a logical requirement into an MT-FSM (we shall see how to do it for LTL in the next subsection), the next important step is to generate an efficient equivalent BTT-FSM. In the worst possible

case one just has to evaluate all the atomic predicates in order to proceed to the next state of a BTT-FSM, so they are preferred to MT-FSMs. What one needs to do is to develop a procedure that takes a multi-transition and returns a BTT. More BTTs can encode the same multi-transition, so one needs to develop some criteria to select the better ones. A natural selection criterion would be to minimize the average amount of computation. For example, if all atomic predicates are equally probable to hold and if an atomic predicate is very expensive to evaluate, then one would select that BTT that places the expensive predicate as deeply as possible, so its evaluation is delayed as much as possible. Based on the above, we believe that the following is an important theoretical problem in runtime monitoring:

Problem: Optimal BTT

Input: A set of atomic predicates a_1, \dots, a_k that hold with probabilities π_1, \dots, π_k and have costs c_1, \dots, c_k , respectively, and a multi-transition $p_1?s_1, \dots, p_n?s_n$ where p_1, \dots, p_n are Boolean formulae over a_1, \dots, a_k .

Output: A BTT implementing the multi-transition that probabilistically minimizes the amount of computation to decide the next state.

The probabilities and the costs in the problem above can be estimated either by static analysis of the source code of the program, or dynamically by first running and measuring the program several times, or by combinations of those. We do not know how to solve this interesting problem yet, but we conjecture the following result that we currently use in our implementation:

Conjecture 7 *If $\pi_1 = \dots = \pi_k$ and $c_1 = \dots = c_k$ then the solution to the problem above is the BTT of minimal size.*

Our current multi-transition to BTT algorithm is exponential in the number of atomic predicates; it simply enumerates all possible BTTs recursively and then selects the one of minimal size. Generating the BTTs takes significantly less time than generating the MT-FSM, so we do not regard it as a practical problem yet. However, it seems to be a challenging theoretical problem.

Example 6 *Consider again the traffic light controller safety property stating that “after green comes yellow”, which was written as $\square(\text{green} \rightarrow (!\text{red} \cup$*

State	MT for non-terminal events	MT for terminal events
1	$\begin{bmatrix} \text{yellow} \setminus / \text{!green} & ? & 1, \\ \text{!yellow} \wedge \text{green} \wedge \text{!red} & ? & 2, \\ \text{!yellow} \wedge \text{green} \wedge \text{red} & ? & \text{false} \end{bmatrix}$	$\begin{bmatrix} \text{yellow} \setminus / \text{!green} & ? & \text{true}, \\ \text{!yellow} \wedge \text{green} & ? & \text{false} \end{bmatrix}$
2	$\begin{bmatrix} \text{yellow} & ? & 1, \\ \text{!yellow} \wedge \text{!red} & ? & 2, \\ \text{!yellow} \wedge \text{red} & ? & \text{false} \end{bmatrix}$	$\begin{bmatrix} \text{yellow} & ? & \text{true}, \\ \text{!yellow} & ? & \text{false} \end{bmatrix}$

Figure 6.2: MT-FSM for the formula $[\text{green} \rightarrow \text{!red} \cup \text{yellow}]$.

`yellow`)) using LTL notation. Since more than one light can be lit at any moment, one should be very careful when expressing this safety property as an MT-FSM or a BTT-FSM.

Let us first express it as an MT-FSM. We need two states, one for the case in which green has not triggered yet the `!red U yellow` part and another for the case when it has. The condition to stay in state 1 is then `yellow \ / !green` and the condition to move to state 2 is `!yellow /\ green /\ !red`. If both a `green` and a `red` are seen then the machine should move to state false. The condition to move from state 2 back to state 1 is `yellow`, while the condition to stay in state 2 is `!yellow /\ !red`; `!yellow /\ red` should also move the machine in its false state. If the event is terminal then a `yellow` would make the reported answer true, i.e., the observed trace is an accepting sequence; if `yellow` is not true, then in state 1 the answer should be the opposite of `green`, while in state 2 the answer should be simply false. This MT-FSM is shown in Figure 6.2.

The interesting aspect of our FSMs is that not all the atomic predicates need to always be evaluated. For example, in state 2 of this MT-FSM the predicate `green` is not needed. A BTT-FSM further reduces the amount of predicates to be evaluated, by enforcing an “evaluate-by-need” policy. Figure 6.3 shows a BTT-FSM implementing the MT-FSM in Figure 6.2.

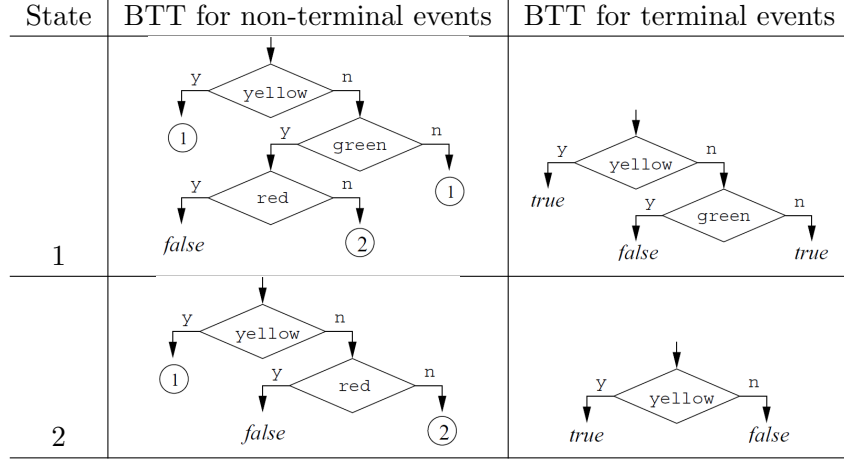


Figure 6.3: A BTT-FSM for the formula $\Box(\text{green} \rightarrow \neg \text{red} \cup \text{yellow})$.

6.2 Multi-Transitions and Binary Transition Trees

Büchi automata cannot be used unchanged as monitors. For the rest of the paper we explore structures suitable for monitoring as well as techniques to transform Büchi automata into such structures. Deterministic multi-transitions (*MT*) and binary-transition trees (*BTTs*) were introduced in [88, 151]. In this section we extend their original definitions with nondeterminism.

Definition 31 Let S and A be sets of **states** and **atomic predicates**, respectively, and let P_A denote the set of **propositions** over atoms in A , using the usual boolean operators. If $\{s_1, s_2, \dots, s_n\} \subseteq S$ and $\{p_1, p_2, \dots, p_n\} \subseteq P_A$, we call the n -tuple $[p_1: s_1, p_2: s_2, \dots, p_n: s_n]$ a **(nondeterministic) multi-transition (MT)** over P_A and S . Let $MT(P_A, S)$ denote the set of MTs over P_A and S .

Intuitively, if a monitor is in a state associated to an *MT* $[p_1: s_1, p_2: s_2, \dots, p_n: s_n]$ then p_1, p_2, \dots, p_n can be regarded as guards allowing the monitor to nondeterministically transit to one of the states s_1, s_2, \dots, s_n .

Definition 32 Maps $\theta: A \rightarrow \{\text{true}, \text{false}\}$ are called **A-events**, or simply **events**. Given an *A-event* θ , we define its **multi-transition extension** as the map $\theta_{MT}: MT(P_A, S) \rightarrow 2^S$, where $\theta_{MT}([p_1: s_1, p_2: s_2, \dots, p_n: s_n]) = \{s_i \mid \theta \models p_i\}$.

The role of A -events is to transmit the monitor information regarding the running program. In any program state, the map θ assigns atomic propositions to *true* iff they hold in that state, otherwise to *false*. Therefore, A -events can be regarded as abstractions of the program states. Moreover, technically speaking, A -events are in a bijective map to P_A . For an MT μ , the set of states $\theta_{MT}(\mu)$ is often called the set of *possible continuations* of μ under θ .

Example 7 If $\mu = [a \vee \neg b: s_1, \neg a \wedge b: s_2, c: s_3]$, and $\theta(a)=\text{true}$, $\theta(b)=\text{false}$, and $\theta(c)=\text{true}$, then the set of possible continuations of μ under θ , $\theta_{MT}(\mu)$, is $\{s_1, s_3\}$.

Definition 33 A (*nondeterministic*) **binary transition tree (BTT)** over A and S is inductively defined as either a set in 2^S or a structure of the form $a ? \beta_1 : \beta_2$, for some atom a and for some binary transition trees β_1 and β_2 . Let $BTT(A, S)$ denote the set of BTTs over the set of states S and atoms A .

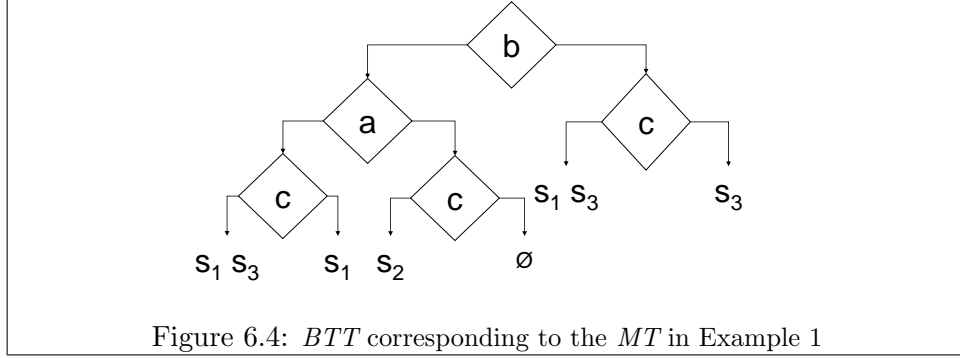
Definition 34 Given an event θ , we define its **binary transition tree extension** as the map $\theta_{BTT}: BTT(A, S) \rightarrow 2^S$, where:

$$\begin{aligned} \theta_{BTT}(Q) &= Q \text{ for any set of states } Q \subseteq S, \\ \theta_{BTT}(a ? \beta_1 : \beta_2) &= \theta_{BTT}(\beta_1) \text{ if } \theta(a) = \text{true}, \text{ and} \\ \theta_{BTT}(a ? \beta_1 : \beta_2) &= \theta_{BTT}(\beta_2) \text{ if } \theta(a) = \text{false}. \end{aligned}$$

Definition 35 A BTT β **implements** an MT μ , written $\beta \models \mu$, iff for any event θ , it is the case that $\theta_{BTT}(\beta) = \theta_{MT}(\mu)$.

Example 8 The BTT $b ? (a ? (c ? s_1 s_3 : s_1) : (c ? s_2 : \emptyset)) : (c ? s_1 s_3 : s_3)$ implements the multi-transition shown in Example 7.

Fig. 6.4 represents this BTT graphically. The right branch of the node labeled with **b** corresponds to the BTT expression $(c ? s_1 s_3 : s_3)$, and similarly for the left branch and every other node. Atomic predicates can be any host programming language boolean expressions. For example, one may be interested if a variable x is positive or if a vector $v[1...100]$ is sorted. Some atomic predicates typically are more expensive to evaluate than others. Since our purpose is to generate *efficient monitors*, we need to take the evaluation costs of atomic predicates into consideration. Moreover, some predicates can hold with higher probability than others; for example, some predicates may be simple “sanity checks”, such as checking whether the output of a



sorting procedure is indeed sorted. We next assume that atomic predicates are given evaluation costs and probabilities to hold. These may be estimated *apriori*, either statically or dynamically.

Definition 36 *If $\varsigma: A \rightarrow \mathcal{R}^+$ and $\pi: A \rightarrow [0, 1]$ are cost and probability functions for events in A , respectively, then let $\gamma_{\varsigma, \pi}: BTT(A, S) \rightarrow \mathcal{R}^+$ defined as:*

$$\gamma_{\varsigma, \pi}(Q) = 0 \text{ for any } Q \subseteq S, \text{ and}$$

$$\gamma_{\varsigma, \pi}(a ? \beta_1 : \beta_2) = \varsigma(a) + \pi(a) * \gamma_{\varsigma, \pi}(\beta_1) + (1 - \pi(a)) * \gamma_{\varsigma, \pi}(\beta_2),$$

be the **expected (evaluation) cost** function on *BTTs* in $BTT(A, S)$.

Example 9 *Given $\varsigma = \{(a, 10), (b, 5), (c, 20)\}$ and $\pi = \{(a, 0.2), (b, 0.5), (c, 0.5)\}$, the expected evaluation cost of the *BTT* defined in Example 2 is 30.*

With the terminology and motivations above, the following problem develops as an interesting and important problem in monitor synthesis:

Problem: Optimal $BTT(A, S)$.

Input: A multi-transition $\mu = [p_1 : s_1, p_2 : s_2, \dots, p_n : s_n]$ with associated cost $\varsigma : A \rightarrow \mathcal{R}^+$ and probability $\pi : A \rightarrow [0, 1]$.

Output: A minimal cost *BTT* β with $\beta \models \mu$.

Binary decision trees (BDTs) and diagrams (BDDs) have been studied as models and data-structures for several problems in artificial intelligence [138] and program verification [41]. Appendix B discusses BDTs and how they relate to *BTTs*. Moret [138] shows that a simpler version of this problem, using BDTs, is NP-hard.

In spite of this result, in general the number of atoms in formulae is relatively small, so it is not impractical to exhaustively search for the optimal

BTT. We next informally describe a backtracking algorithm that we are currently using in our implementation to compute the minimal cost *BTT* by exhaustive search. Start with the sequence of all atoms in A . Pick one atom, say a , and make two recursive calls to this procedure, each assuming one boolean assignment to a . In each call, pass the remaining sequence of atoms to test, and simplify the set of propositions in the multi-transition according to the value of a . The product of the *BTTs* is taken when the recursive calls return in order to compute all *BTTs* starting with a . This procedure repeats until no atom is left in the sequence. We select the minimal cost *BTT* amongst all computed.

6.3 Binary Transition Tree Finite State Machines

We next define an automata-like structure, formalizing the desired concept of an *effective runtime monitor*. The transitions of each state are all-together encoded by a *BTT*, in practice the statistically optimal one, in order for the monitor to efficiently transit as events take place in the monitored program. Violations occur when one cannot further transit to any state for any event. A special state, called *neverViolate*, will denote a configuration in which one can no longer detect a violation, so one can stop the monitoring session if this state is reached.

Definition 37 A *binary transition tree finite state machine (BTT-FSM)* is a tuple $\langle A, S, btt, S_0 \rangle$, where A is a set of atoms, S is a set of states potentially including a special state called “*neverViolate*”, btt is a map associating a *BTT* in $BTT(A, S)$ to each state in S where $btt(\text{neverViolate}) = \{\text{neverViolate}\}$ when $\text{neverViolate} \in S$, and $S_0 \subseteq S$ is a subset of initial states.

Definition 38 Let $\langle A, S, btt, S_0 \rangle$ be a *BTT-FSM*. For an event θ and $Q, Q' \subseteq S$, we write $Q \xrightarrow{\theta} Q'$ and call it a **transition between sets of states**, whenever $Q' = \bigcup_{s \in Q} \theta_{BTT}(btt(s))$. A **trace of events** $\theta_1 \theta_2 \dots \theta_j$ generates a **sequence of transitions** $Q_0 \xrightarrow{\theta_1} Q_1 \xrightarrow{\theta_2} \dots \xrightarrow{\theta_j} Q_j$ in the *BTT-FSM*, where $Q_0 = S_0$ and $Q_i \xrightarrow{\theta_{i+1}} Q_{i+1}$, for all $0 \leq i < j$. The trace is **rejecting** iff $Q_j = \{\}$.

Note that no finite extension of a trace $\theta_1 \theta_2 \dots \theta_j$ will be **rejected** if $\text{neverViolate} \in Q_j$. The state *neverViolate* denotes a configuration in which violations can no longer be detected for any finite trace extension. This means that the set Q_k will not be empty, for any $k > j$, when *neverViolate*

$\in Q_j$. Therefore, the monitoring session can stop at event j if *neverViolate* $\in Q_j$, because we are only interested in violations of requirements.

Exercises

Exercise 8 *Give pseudo-code for the problem of generating a minimal BTT for a given MT (described in Section 6.1.2 and right after Definition 36).*

Exercise^{*} 9 *Prove or disprove Conjecture 7.*

Chapter 7

Monitor Synthesis: Extended Regular Expressions (ERE)

bad vs. good prefixes — cite Vardi

7.1 Monitoring ERE Safety Needs Non-Elementary Space

Extended regular expressions (EREs) add complementation (\neg) to regular expressions (REs). Complementation can be handy when defining safety properties, because it allows us to say both what should never happen as well as what could happen. In particular, in the context of monitoring EREs, one can switch between the expression of bad prefixes of a safety property and that of good prefixes by just applying a complement operator.

In this section we show that any monitor for safety properties expressed as EREs requires non-elementary space. More precisely, for a given $n \in \mathbb{N}$, we build an ERE of size $O(n^3)$ such that its language is prefix closed and any monitor for its language requires space

$$2^{2^{\cdot^{2^n}}} \quad \Bigg] \quad n \text{ nested power of 2 operations}$$

discuss that this also implies non-elementary time complexity

7.1.1 Discussion and Relevance of the Lower-Bound Result

move some of the discussion below to Chapter 6

Since regular expressions (REs) and deterministic (DFA) and non-deterministic (NFA) finite-state automata are enumerable structures while the set of safety properties is in bijection with the set of real numbers, there are many safety properties that cannot be expressed using REs or automata (or any other formalism whose objects are enumerable). Nevertheless, there are many safety properties of interest that can be expressed as REs or automata. Safety properties over finite or infinite traces can be expressed as REs in at least two different ways:

1. Use an RE to express the language of its bad prefixes; or
2. Use an RE to express the language of its (good) prefixes.

In the first case, the RE captures the finite-trace behaviors that should never happen, while in the second the RE captures the ones that could possibly happen.

Obviously, not all REs correspond to safety properties in one or the other of the two cases above. For example, the RE $(0 \cdot 1)^+$ cannot express the bad prefixes of a safety property, because “01” is a bad prefix while “010” is not. In order for an RE to express the bad prefixes of a safety property, it should have the property that once w is in its language, then all ww' for any w' should also be in its language. The RE $(0 \cdot 1)^+$ cannot express the good prefixes of a safety property either: “0101” is a good prefix while 010 is not. The language of an RE must be prefix closed in order to express the good prefixes of a safety property.

In both cases above, monitoring the safety property can be done very efficiently (linearly in the size of the RE, both space-wise and time-wise) by first translating its corresponding RE into an NFA, for example using a technique such as Thompson’s [174], and then doing one of the following:

1. In the first case, simulate the NFA-to-DFA construction on the fly as events are received. The state of the monitor is therefore a set of states of the NFA. At start, that set contains only the initial state of the NFA. For each new event, construct the next set by collecting all the NFA states that can be reached via the received event from any of the existing states in the set. If a final state is reached then report

violation of the property: bad prefix found. Since the final states in the NFA symbolize the reach of a bad prefix and since bad prefixes have “no future” in a safety property, the NFA associated to the original RA can be optimized (in case it is not already optimal directly from its construction) by removing any edges out of its final states.

2. In the second case, the monitor works the same way as in the first case, but checking at each time that the monitor state (also a set of NFA states) contains at least one final state of the NFA; if that is not the case, then report violation: prefix found which is not good. If one is willing to pay the exponential price and determinize the NFA of good prefixes, then one can further optimize the resulting DFA (in case it is not already optimized by construction) by collapsing all its non-final states into a “dead-end” state: indeed, the reachability of a non-final state signifies the reachability of a bad prefix, which has “no future”. It is not clear whether or how to optimize the NFA of good prefixes using the additional info that it is a safety property.

almost all the discussion above can go to Chapter 6.

Extended regular expressions (EREs) add complementation (\neg) to REs. Meyer and Stockmeyer [170, 171] showed that EREs can express languages non-elementarily more compactly than REs. In other words, for any constant $k \geq 1$, one can find EREs of large enough size $n \in \mathbb{N}$ for which there is no RE having the same language of size less than $2^{2^{\dots^{2^n}}}$, with k nested power operations. Meyer and Stockmeyer [170, 171] showed that several other problems concerning EREs are also non-elementary, including: the equivalence of EREs, the emptiness of the language of an ERE (and implicitly the emptiness of the complement of the language of an ERE), the automata generation (NFA or DFA) from an ERE, etc. Note that it is straightforward to generate potentially non-elementarily large automata from EREs. All one needs to do is to iteratively apply NFA-to-DFA transformations for EREs under complement operators and then complement the resulting DFAs (by complementing their final states). Since each NFA-to-DFA transformation may lead to an exponential explosion on the number of states and since the ERE can have arbitrarily many nested complement operators, the resulting NFA or DFA can be non-elementarily larger than the original ERE.

As already mentioned above, if we allow complementation then we can easily switch from an expression defining the bad prefixes of a safety property

to one defining its good prefixes, and backwards, by applying a complement operator. Therefore, from here on, when we say that an ERE expresses a safety property, without any loss of generality we assume that it defines the good prefixes of the safety property; in particular, we assume that its language is prefix closed. Clearly, if one can afford to generate an automaton from the ERE expressing a safety property, then one can and probably should use that automaton as a monitor for the safety property. However, since such an automaton can be enormous compared to the size of the original ERE, a natural question to ask is whether one can generate monitors for safety properties expressed as EREs that need less than non-elementary space in the size of the original ERE. We next give a negative answer to this question.

Let us first discuss our subsequent lower bound result from a more conceptual perspective. Notice that *synchronous monitoring*, that is, the monitoring process where an error is detected as soon as it appears, is harder than checking for satisfiability (or non-emptiness); indeed, if a formula in a particular formalism is not satisfiable (or it has an empty language), then a synchronous monitor should detect that before the first event is observed. We refer the interested reader to [150] for a discussion on various types of monitoring, including synchronous versus asynchronous monitoring. Synchronous monitors need to either directly (e.g., by accumulating logical constraints while verifying their consistency) or indirectly (e.g., by generating statically an automaton or a structure containing all possible future behaviors) check for satisfiability (or non-emptiness) of the remaining requirements as events are observed online. Unfortunately, this is an expensive process that may be desired to be avoided, at the expense of delaying the detection of violations. For example, the rewriting based monitoring approach for LTL in [150] delays the detection of violations of LTL formulae of the form “(next φ) and (next $\neg\varphi$)” for one more event, to avoid invoking an expensive satisfiability checker for LTL but to instead invoke a propositional satisfiability checker which is less expensive in practice; this is closely related to the notion of “informative prefixes” in [123] that “tell all the story”. Our subsequent lower-bound result states that any monitor for safety properties expressed using EREs, *synchronous or asynchronous*, requires non-elementary space.

Let us now clarify that our lower bound result is not a consequence of the lower-bound result by Meyer and Stockmeyer in [170] (see [171] for a proof of that result). A first reason is that neither the ERE constructed in [171, 170] nor its complement is prefix closed. Indeed, we here focus on a subset of

EREs, rather than arbitrary EREs, namely those whose languages are prefix closed, so they express good prefixes of safety properties. Supposing that one could modify the “hard” ERE in [171, 170] whose complement non-emptiness requires non-elementary space into one whose language is prefix-closed and whose size is linear in the size of the original one, the fact that synchronous monitoring of EREs is harder than checking for emptiness does not necessarily imply that monitoring that hard ERE requires non-elementary space. In fact, monitoring that particular ERE requires constant time to process each event, because it is equivalent with an automaton of one state – it takes, however, non-elementary space to compute that one state automaton. What it says is therefore that the *initialization step of ERE-safety synchronous monitoring* requires, in the worst case, non-elementary space.

Interestingly, if one could modify the results in [171, 170] to hold for prefix-closed EREs, including especially the result stating that finding an RE equivalent to an ERE is a non-elementary problem, then one could show that *synchronous monitoring* of safety properties expressed as EREs requires non-elementary space! Indeed, supposing that one had for any ERE a monitor that takes only elementary space in its worst-case monitoring scenario, then one could use that monitor to generate a DFA for the ERE as follows: start with the initial state of the monitor and then discover and store new states of the monitor by feeding arbitrary (but finite in number) events to each state of the monitor until no new state is discovered. This closure operation takes as much time and space as the number of states the monitor reach; since by assumption the monitor needs “only” elementary space to store its state in any scenario, we deduce that the obtained automaton has size elementary in the size of the ERE (and it also takes elementary time and space to generate it). In other words, we could find an elementary algorithm to associate to any (prefix closed) ERE an equivalent RE, contradicting the non-elementary lower bound in [171, 170] (again, supposing that the lower-bound results in [171, 170] could be modified for prefix-closed EREs).

Unfortunately, it is not that clear how to reduce the non-elementary problems in [171, 170] to *asynchronous monitoring*, and thus to conclude that asynchronous monitoring also requires non-elementary space. That is because a “smart” asynchronous monitor may in principle collapse states (when regarded as an automaton as in the construction above) in rather unexpected ways, just because it “knows” that eventually an error may be reported anyway if observation continues indefinitely from that state on; in other words, states with the property that “eventually violation detected

in the future” may be collapsed as equivalent by an asynchronous monitor. This way, the DFA extracted from a monitor for a safety property expressed as an ERE may be significantly smaller than the DFA corresponding to the ERE (and obviously, it may have a different language).

Our next result shows that asynchronous monitoring of ERE-safety also requires non-elementary space, which is a more general lower-bound result than the space non-elementarity of synchronous monitoring. Moreover, it gives an alternative proof of the non-elementary lower-bounds by Stockmeyer and Meyer [171, 170], because automata corresponding to safety-defining EREs are just special cases of monitors, so they must also take non-elementary space. Moreover, we improve the results in [171, 170] by showing that their lower-bounds also hold for a *subset of EREs*, namely those corresponding to safety properties.

Summarizing the discussion above, we believe that the main contributions of our subsequent lower-bound result are the following:

1. We show that asynchronous monitoring already requires non-elementary space, same as synchronous monitoring. For example, an ERE monitoring algorithm was presented by Roġu and Viswanathan in [147], which “rewrites” or “derives” the ERE by each letter in the input word; the derivation process consists of some straightforward rewrite rules, some for expanding the ERE others for contracting it via simplifications. No comprehensive and expensive check for emptiness on the resulting ERE is performed, except what is done by the simplification rules (for example $\emptyset \cdot R \rightarrow \emptyset$ and $\epsilon \cdot R \rightarrow R$, etc.). A check for emptiness can and should be eventually performed (for example, a check for emptiness can be done periodically, say every 10^x events for some convenient x). This gives us an asynchronous ERE monitoring algorithm, which, unlike the simplistic NFA/DFA generation algorithm, does not pay upfront the potentially non-elementary worst-case penalty! However, our subsequent lower bound result tells that there is a worst-case scenario in which one cannot avoid the non-elementary space required to store the continuously changing ERE if one wants to correctly eventually detect violations of the original ERE. And that is the case for any synchronous or asynchronous monitoring algorithm for safety properties expressed as EREs.
2. We propose a different technique to prove non-elementary lower-bounds, fundamentally different from the one in [171]. The technique in [171] is

based on diagonalization arguments and encodings of accepting Turing machine computations as finite trace words. Our technique is inspired from an idea by Chandra, Kozen and Stockmeyer [32] introduced to show the power of alternation and then used by several authors to prove exponential lower bounds [119, 120, 147, 150]. At our knowledge, the use of such a technique to show non-elementary lower bounds is novel. The idea of the technique in [32] is to define expressions having as languages $L_n = \{w^{(1)}\#w^{(2)}\#\dots\#w^{(k)}\$w \mid w^{(1)}, w^{(2)}, \dots, w^{(k)}, w \in \{0, 1\}^n, (\exists 1 \leq i \leq k) w^{(i)} = w\}$. Our idea is to define, using EREs, words of the form $X_n\$X'_n$, where X_n and X'_n are *n-deep nested sets* starting with elements in $\{0, 1\}^n$ (i.e., sets of sets of ... of sets of elements in $\{0, 1\}^n$, with n power set operations), such that X'_n is *n-nested included* in X_n , where *i-nested inclusion* is standard inclusion when $i = 1$ and, if $i > 1$, then it is defined inductively as: X'_i is *i-nested included* in X_i iff for each $X'_{i-1} \in X'_i$, there is some $X_{i-1} \in X_i$ such that X'_{i-1} is $(i - 1)$ -nested included in X_{i-1} .

One more observation is in place before we move on to the technical details. It is known that the *membership problem* for EREs, testing whether a word w of size m is in the language of an ERE of size n , is polynomial in m and n . For example, the classic algorithm by Hopcroft and Ullman [104] runs in space $O(m^2 \cdot n)$ and time $O(m^3 \cdot n)$; slightly improved algorithms have been proposed by several authors [99, 177, 178, 179, 122, 108], reducing space requirements to $O(m^2 \cdot k + m \cdot n)$ and time to $O(m^3 \cdot k + m^2 \cdot n)$ or worse, where k is the number of complement operators in the ERE; a recent ERE membership algorithm was proposed by the author in [153], which runs in space $O(m \cdot \log m \cdot 2^n)$ and time $O(m^2 \cdot \log m \cdot 2^n)$ when $m > 2^n$. These algorithms appear to be efficient, because they are polynomial or simply exponential in the ERE, so one may think that one may device an elementary ERE-safety monitoring algorithm as follows: store the trace of events and at each newly received event invoke one of these “efficient” ERE membership algorithms. While this algorithm will indeed be elementary in the size of the ERE *and* the size of the trace, our lower bound result says that it will, in fact, be *non-elementary in the size of only the ERE!* In other words, for a carefully chosen “hard” ERE of size n , there are finite traces of large enough size m so that checking their membership is a problem which is non-elementary in n ; this will indeed happen when m is non-elementarily larger than n .

7.1.2 The Lower-Bound Result

EREs define languages by inductively applying *union* ($+$), *concatenation* (\cdot), *Kleene Closure* (\star), *intersection* (\cap), and *complementation* (\neg). The language of an ERE R , say $\mathcal{L}(R)$, is defined inductively as follows, where $s \in \Sigma$:

- $\mathcal{L}(\emptyset) = \emptyset$,
- $\mathcal{L}(\epsilon) = \{\epsilon\}$,
- $\mathcal{L}(s) = \{s\}$,
- $\mathcal{L}(R_1 + R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$,
- $\mathcal{L}(R_1 \cdot R_2) = \mathcal{L}(R_1) \cdot \mathcal{L}(R_2)$,
- $\mathcal{L}(R^\star) = (\mathcal{L}(R))^\star$,
- $\mathcal{L}(R_1 \cap R_2) = \mathcal{L}(R_1) \cap \mathcal{L}(R_2)$,
- $\mathcal{L}(\neg R) = \neg \mathcal{L}(R)$.

If R does not contain \neg and \cap then it is a *regular expression* (RE). By applying De Morgan's law $R_1 \cap R_2 \equiv \neg(\neg R_1 + \neg R_2)$, EREs can be linearly (in both time and size) translated into equivalent EREs without intersection; therefore, intersection of EREs is just syntactic sugar. The *size* of an ERE is the total number of occurrences of letters and composition operators ($+$, \cdot , \star , and \neg) that it contains. In what follows we assume that Σ is finite. For notational simplicity, in what follows we let Σ also denote the RE $s_1 + s_2 + \dots + s_n$ where $\Sigma = \{s_1, s_2, \dots, s_n\}$ and let Σ^\star denote both the language $\{s_1, s_2, \dots, s_n\}^\star$ and the RE $(s_1 + s_2 + \dots + s_n)^\star$.

For $n \in \mathbb{N}$, let us define inductively the following alphabets and languages:

- $\Sigma_0 = \{0, 1\}$ and $\Psi_0 = \{0, 1\}^n$, and
- $\Sigma_i = \Sigma_{i-1} \cup \{\#_i\}$ and $\Psi_i = \{\#_i \#_i\} \cup (\{\#_i\} \cdot \Psi_{i-1})^+ \cdot \{\#_i\}$, for all $1 \leq i \leq n$.

In the above, $\#_i$ are n fresh letters. The intuition for the languages Ψ_i above is to encode nested sets of depth i that contain sets of words of n bits at their deepest level. The symbols $\#_i$ play the role of markers separating the elements of such sets. For example, the word $\#_2 \#_1 \#_1 \#_2 \#_1 01 \#_1 10 \#_1 \#_2 \#_1 \#_1 \#_2$

encodes $\{\{\}, \{01, 10\}, \{\}\}$, that is, the set $\{\{\}, \{01, 10\}\}$; since the multiplicity and order of elements in sets are irrelevant, the same set can have (infinitely) many different encodings. Formally, let us define the following *set* functions associating to encodings in Ψ_i corresponding nested sets:

- $set_0 : \Psi_0 \rightarrow \{0, 1\}^n$ is the identity function on Ψ_0 , i.e., $set_0(w) = w$;
- $set_i : \Psi_i \rightarrow \mathcal{P}^i(\{0, 1\}^n)$, where \mathcal{P}^i is the power set operator applied i times, $set_i(\#_i \#_i) = \{\}$, $set_i(\#_i X_{i-1} \#_i) = \{set_{i-1}(X_{i-1})\}$, and $set_i(\#_i X_{i-1} X_i) = \{set_{i-1}(X_{i-1})\} \cup set_i(X_i)$, for all $1 \leq i \leq n$, $X_{i-1} \in \Psi_{i-1}$, and $X_i \in \Psi_i$.

Note that $|set_0(\Psi_0)| = 2^n$ and $set_i(\Psi_i) = \mathcal{P}(set_{i-1}(\Psi_{i-1}))$ for all $1 \leq i \leq n$; therefore, $|set_i(\Psi_i)| = 2^{2^{i-1} \cdot 2^n}$ for all $1 \leq i \leq n$, with $i + 1$ nested power operations.

Let us define *nested-inclusion* relations $_ \subseteq_i _ : \mathcal{P}^i(\{0, 1\}^n) \times \mathcal{P}^i(\{0, 1\}^n)$ for $0 \leq i \leq n$ and *nested-membership* relations $_ \in_i _ : \mathcal{P}^{i-1}(\{0, 1\}^n) \times \mathcal{P}^i(\{0, 1\}^n)$ for $1 \leq i \leq n$ as follows:

- $_ \subseteq_0 _$ is the identity on $\{0, 1\}^n$ and $_ \subseteq_1 _$ is $_ \subseteq _ : \mathcal{P}(\{0, 1\}^n) \times \mathcal{P}(\{0, 1\}^n)$,
- $_ \in_1 _$ is $_ \in _ : \{0, 1\}^n \times \mathcal{P}(\{0, 1\}^n)$, and for $1 < i$,
- $S_{i-1} \in S_i$ iff there is some $S'_{i-1} \in S_i$ such that $S_{i-1} \subseteq S'_{i-1}$, and
- $S_i \subseteq S'_i$ iff $S_{i-1} \in S'_i$ for each $S_{i-1} \in S_i$.

For example, if $n = 2$ then $\{\{00, 01\}, \{01, 10\}, \{11\}\} \subseteq_2 \{\{00, 01, 10\}, \{00, 11\}\}$ because $\{00, 01\}, \{01, 10\} \subseteq_1 \{00, 01, 10\}$ and $\{11\} \subseteq_1 \{00, 11\}$.

We can now define Σ as $\Sigma_n \cup \{\$\}$ and the infinite trace property P_n^ω over Σ :

$$(\epsilon \cup (\Sigma_n^* \cup \{X_n \$ X'_n \mid X_n, X'_n \in \Psi_n \text{ and } set_n(X'_n) \subseteq_n set_n(X_n)\}) \cdot \{\$\}) \cdot \Sigma_n^\omega.$$

An infinite trace in P_n^ω therefore contains at most two $\$$ letters and infinitely many letters in Σ_n . There are no restrictions on the appearance of the letters in Σ_n when there is no $\$$ letter or when there is precisely one $\$$ letter. However, if the infinite trace contains precisely two $\$$ letters, that is, if it has the form $w \$ w' \$ u$ for some $w, w' \in \Sigma_n^*$ and some $u \in \Sigma_n^\omega$, then w and w' must be in Ψ_n and the nested set corresponding to w must nested-include the nested set corresponding to w' ; there are no restrictions on u .

Proposition 15 $P_n^\omega \in \text{Safety}^\omega$.

Proof: There are two cases to analyze for an infinite trace that is not in P_n^ω : when it contains more than two \$ letter, or when it has the form $w\$w'\u with $w, w' \in \Sigma_n^*$ and $u \in \Sigma_n^\omega$, but it is not the case that $w, w' \in \Psi_n$ and $\text{set}_n(w') \subseteq_n \text{set}_n(w)$. In the first case, we can pick the first prefix of the infinite trace containing three \$ letters in total; clearly, this finite trace prefix cannot be continued into any acceptable infinite trace. In the second case, since there are no restrictions on the letters in u , we can easily see that the prefix $w\$w'\$$ is already a violation threshold: there is no $u' \in \Sigma_n^\omega$ such that $w\$w'\$u' \in P_n^\omega$. \square

The bijection in the proof of Theorem 3 associates to each infinite-trace safety property a persistent finite-trace safety property by taking its prefixes. Let P_n be the persistent finite-trace safety property $\text{prefixes}(P_n^\omega)$ corresponding to P_n^ω . It is easy to see that P_n is the property

$$\Sigma_n^* \cup \Sigma_n^* \cdot \{\$ \} \cdot \Sigma_n^* \cup \{X_n\$X'_n \mid X_n, X'_n \in \Psi_n \text{ and } \text{set}_n(X'_n) \subseteq_n \text{set}_n(X_n)\} \cdot \{\$ \} \cdot \Sigma_n^*.$$

Note that monitoring P_n^ω is the same as monitoring P_n : in both cases, besides the capability to checking whether there are more than two \$ letters, which is trivial, the monitor needs to store sufficient information about the nested set corresponding to X_n , so that, when the first \$ is seen, to be able to check whether it nested-includes the set corresponding to the upcoming, yet unknown X'_n .

Theorem 8 *Any synchronous or asynchronous monitor for P_n or P_n^ω needs space non-elementary in n , namely $\Omega(2^{2^{\cdot^{2^n}}})$, with n nested power operations.*

Proof: Suppose that M is a monitor for P_n or P_n^ω and suppose that, during a monitoring session, it reads the prefix $X_n \in \Psi_n$. Regardless of how M is defined or implemented, in particular regardless of whether it reports violations synchronously or asynchronously, when the first \$ event is encountered, the state of M must contain enough information to sooner or later be able to decide whether the set $\text{set}_n(X'_n)$ corresponding to *any* upcoming (unknown at the time the \$ is observed) sequence X'_n is nested-included in $\text{set}_n(X_n)$. Since $\text{set}_n(X'_n)$ can in particular be equal to $\text{set}_n(X_n)$, and since once the second \$ event is observed there is no further event that may bring new knowledge to the monitor, we deduce that M must be able

to distinguish any two different sets in $set_n(\Psi_n)$ when the first \$ event is encountered, that is, M 's states must be different after reading words in Ψ_n whose corresponding nested sets are different. Therefore, M must be able to distinguish $|set_n(\Psi_n)|$ different possibilities. Since one needs $\Omega(\log N)$ space to distinguish among N different situations (one label for each), we conclude that M needs space $\Omega(\log(|set_n(\Psi_n)|))$, that is, $\Omega(2^{2^n})$ with n nested power operations. Hence, any monitor for P_n needs non-elementarily large space in n . \square

We next show how to construct an ERE polynomial in size with n whose language is precisely P_n .

Theorem 9 *There is an ERE of size $O(n^3)$ whose language is P_n .*

Proof: Note that P_n is a union of three languages, the first two being trivial to express as languages of corresponding REs. As expected, the difficult part is to associate an ERE to the language

$$\{X_n \$ X'_n \mid X_n, X'_n \in \Psi_n \text{ and } set_n(X'_n) \subseteq_n set_n(X_n)\}.$$

Note that so far we did not need complementation. The property above can, however, be expressed as an ERE of size $O(n^3)$ using $O(n)$ nested complement operators. The idea is to define iteratively a sequence of EREs \mathbb{K}_i for $0 \leq i \leq n$ whose languages contain words $X_i w \$ w' X'_i$ with $set_i(X'_i) \subseteq_i set(X_i)$, which are contiguous fragments of desired words $X_n \$ X'_n$. Then \mathbb{K}_n would be the language that we are looking for. To define \mathbb{K}_i , we observe that the nested-inclusion $S'_i \subseteq_i S_i$ is equivalent to: there is no $S'_{i-1} \in S'_i$ such that it is not the case that we can find some $S_{i-1} \in S_i$ such that $S'_{i-1} \subseteq_{i-1} S_{i-1}$. This crucial observation will allow us to define \mathbb{K}_i in terms of \mathbb{K}_{i-1} . We next develop the technical details.

Let us first define regular patterns corresponding to each of the languages Ψ_i for $0 \leq i \leq n$; to avoid introducing new names, we ambiguously let the corresponding regular expressions have the same names as their languages:

- Let $\Psi_0 = (0 + 1)^n$, where for an RE, r^n is $r \cdot r \cdot \dots \cdot r$ (n times); and
- Let $\Psi_i = \#_i \cdot \#_i + (\#_i \cdot \Psi_{i-1})^+ \cdot \#_i$ for all $1 \leq i \leq n$.

Iteratively eliminating the Ψ_{i-1} regular expressions from the right-hand-sides, we eventually obtain $n + 1$ regular patters, each of size $O(n)$ (the size of Ψ_0 as a regular expression is $O(n)$ and each Ψ_i adds a constant size to that of Ψ_{i-1}).

Next we define REs for the languages $\text{prefixes}(\Psi_i)$ and $\text{suffixes}(\Psi_i)$ of prefixes and respectively suffixes of words in Ψ_i , for all $0 \leq i \leq n$. We only discuss the prefix closure languages; the suffix closures are dual. The prefix closures can be defined relatively easily inductively as follows:

- $\text{prefixes}(\Psi_0) = \bigcup_{k=0}^n \{0, 1\}^k$, and
- $\text{prefixes}(\Psi_i) = \{\epsilon, \#_i \#_i\} \cup \{\#_i\} \cdot \text{prefixes}(\Psi_{i-1}) \cup (\{\#_i\} \cdot \Psi_{i-1})^+ \cup (\{\#_i\} \cdot \Psi_{i-1})^+ \cdot \{\#_i\} \cdot \text{prefixes}(\Psi_{i-1})$
 $= \{\#_i \#_i\} \cup (\{\#_i\} \cdot \Psi_{i-1})^* \cdot (\{\epsilon\} \cup \{\#_i\} \cdot \text{prefixes}(\Psi_{i-1})).$

These languages can be expressed with the following REs; as before, we ambiguously use the same names for the corresponding REs:

- Let $\text{prefixes}(\Psi_0) = \epsilon + (0 + 1) + (0 + 1)^2 + \dots + (0 + 1)^n = (\epsilon + 0 + 1)^n$; and
- Let $\text{prefixes}(\Psi_i) = \#_i \cdot \#_i + (\#_i \cdot \Psi_{i-1})^* \cdot (\epsilon + \#_i \cdot \text{prefixes}(\Psi_{i-1})).$

Iteratively eliminating the REs $\text{prefixes}(\Psi_{i-1})$ from the right-hand-sides, we eventually obtain $n + 1$ REs, each of size $O(i^2 + n)$ (the size of $\text{prefixes}(\Psi_0)$ as an RE is $O(n)$ and each $\text{prefixes}(\Psi_i)$ adds size $O(i)$ to that of $\text{prefixes}(\Psi_{i-1})$). Dually,

- Let $\text{suffixes}(\Psi_0) = \epsilon + (0 + 1) + (0 + 1)^2 + \dots + (0 + 1)^n = (\epsilon + 0 + 1)^n$; and
- Let $\text{suffixes}(\Psi_i) = \#_i \cdot \#_i + (\epsilon + \text{suffixes}(\Psi_{i-1}) \cdot \#_i) \cdot (\Psi_{i-1} \cdot \#_i)^*.$

We next define REs L_i and R_i for $0 \leq i \leq n$ whose languages contain the contiguous fragments of words in Ψ_n that are allowed to appear to the left and to the right of $\$,$ respectively, so that words in $\mathcal{L}(L_i)$ start with $\#_i$ and words in $\mathcal{L}(R_i)$ end with $\#_i$. Let us also assume by convention that $L_{n+1} = R_{n+1} = \epsilon$ (the RE whose language contains only the empty word). It is easy to see that L_i and R_i can be defined as follows:

- Let $L_i = \#_i \cdot \Sigma_n^* \cap \text{suffixes}(\Psi_n)$, and
- Let $R_i = \Sigma_n^* \cdot \#_i \cap \text{prefixes}(\Psi_n).$

Note that words in L_i and R_i may not necessarily start or end with a word in Ψ_i : indeed, the $\#_i$ that may start or end L_i or R_i could very well be followed or preceded, respectively, by a $\#_{i+1}$ or, if $i = n$, by $\$$.

Let us also define the EREs \bar{L}_i and \bar{R}_i whose languages are included in those of L_i and R_i , respectively, and whose words start or end with words in Ψ_i :

- Let $\bar{L}_i = \Psi_i \cdot \Sigma_n^* \cap \text{suffixes}(\Psi_n)$, and
- Let $\bar{R}_i = \Sigma_n^* \cdot \Psi_i \cap \text{prefixes}(\Psi_n)$.

It is not difficult to see that $\bar{L}_i = \Psi_i \cdot L_{i+1}$ and $\bar{R}_i = R_{i+1} \cdot \Psi_i$. Note that the sizes of L_i , R_i , \bar{L}_i and \bar{R}_i are $O(n^2)$.

Let us now define the EREs \mathbb{K}_i for $0 \leq i \leq n$ as follows:

- $\mathbb{K}_0 = \bar{L}_0 \cdot \$ \cdot \bar{R}_0 \cap \bigcap_{k=0}^{n-1} (\Sigma_0^k \cdot 0 \cdot \Sigma^\star \cdot 0 \cdot \Sigma_0^{n-k-1} + \Sigma_0^k \cdot 1 \cdot \Sigma^\star \cdot 1 \cdot \Sigma_0^{n-k-1})$,
and
- $\mathbb{K}_i = \bar{L}_i \cdot \$ \cdot \bar{R}_i \cap \neg((\neg((\#_i \cdot \Psi_{i-1})^\star \cdot \#_i \cdot \mathbb{K}_{i-1}) \cap L_i \cdot \$ \cdot R_{i-1}) \cdot (\#_i \cdot \Psi_{i-1})^\star \cdot \#_i)$.

We next show by induction on i that $\mathcal{L}(\mathbb{K}_i)$ is the language

$$\{X_i w X'_i \mid X_i, X'_i \in \Psi_i, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_{i+1}), \text{set}_i(X'_i) \subseteq_i \text{set}_i(X_i)\}.$$

It is easy to see that $\mathcal{L}(\mathbb{K}_0) = \{X_0 w X_0 \mid X_0 \in \Psi_0, w \in \mathcal{L}(L_1 \cdot \$ \cdot R_1)$, because the large conjunct in \mathbb{K}_0 states that the words formed with the first n letters and with the last ones, respectively, are equal and in Ψ_0 , and because $\bar{L}_0 \cdot \$ \cdot \bar{R}_0 = \Psi_0 \cdot L_1 \cdot \$ \cdot R_1 \cdot \Psi_0$ and \subseteq_0 is the identity on $\{0, 1\}^n$. For the inductive step, let us now assume that for some arbitrary $1 \leq i < n$, $\mathcal{L}(\mathbb{K}_{i-1})$ is the language

$$\{X_{i-1} w X'_{i-1} \mid X_{i-1}, X'_{i-1} \in \Psi_{i-1}, w \in \mathcal{L}(L_i \cdot \$ \cdot R_i), \text{set}_{i-1}(X'_{i-1}) \subseteq_{i-1} \text{set}_{i-1}(X_{i-1})\}.$$

Then we can easily show that $\mathcal{L}((\#_i \cdot \Psi_{i-1})^\star \cdot \#_i \cdot \mathbb{K}_{i-1})$ is the language

$$\{X_i w X'_{i-1} \mid X_i \in \Psi_i, X'_{i-1} \in \Psi_{i-1}, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_i), \text{set}_{i-1}(X'_{i-1}) \subseteq_{i-1} \text{set}_i(X_i)\}.$$

Then we can show that $\mathcal{L}(\neg((\#_i \cdot \Psi_{i-1})^\star \cdot \#_i \cdot \mathbb{K}_{i-1}) \cap \bar{L}_i \cdot \$ \cdot \bar{R}_{i-1})$ is

$$\{X_i w X'_{i-1} \mid X_i \in \Psi_i, X'_{i-1} \in \Psi_{i-1}, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_i), \text{set}_{i-1}(X'_{i-1}) \not\subseteq_{i-1} \text{set}_i(X_i)\}.$$

Now we can show that $\mathcal{L}(\neg((\#_i \cdot \Psi_{i-1})^\star \cdot \#_i \cdot \mathbb{K}_{i-1}) \cap \bar{L}_i \cdot \$ \cdot \bar{R}_{i-1}) \cdot (\#_i \cdot \Psi_{i-1})^\star \cdot \#_i$ is the language

$$\{X_i w X'_i \mid X_i, X'_i \in \Psi_i, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_{i+1}), \text{set}_i(X'_i) \not\subseteq_{i-1} \text{set}_i(X_i)\}.$$

Finally, we are now able to show that $\mathcal{L}(\mathbb{K}_i)$, that is,

$$\mathcal{L}(\bar{L}_i \cdot \$ \cdot \bar{R}_i \cap \neg((\neg((\#_i \cdot \Psi_{i-1})^\star \cdot \#_i \cdot \mathbb{K}_{i-1}) \cap L_i \cdot \$ \cdot R_{i-1}) \cdot (\#_i \cdot \Psi_{i-1})^\star \cdot \#_i))$$

is the language

$$\{X_i w X'_i \mid X_i, X'_i \in \Psi_i, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_{i+1}), \text{set}_i(X'_i) \sqsubseteq_i \text{set}_i(X_i)\}.$$

Since $L_{n+1} = R_{n+1} = \epsilon$, it follows that

$$\mathcal{L}(\mathbb{K}_n) = \{X_n \$ X'_n \mid X_n, X'_n \in \Psi_n, \text{set}_i(X'_i) \sqsubseteq_i \text{set}_i(X_i)\}.$$

The size of \mathbb{K}_n is $O(n^3)$.

We can now show that the language of the ERE of size $O(n^3)$

$$(\epsilon + (\Sigma_n^* + \mathbb{K}_n) \cdot \$) \cdot \Sigma_n^*$$

is indeed P_n .

We can now formulate our space lower-bound result for monitoring ERE-safety as a corollary of the two results above.

Corollary 1 *For any $n \in \mathbb{N}$, there is some safety property whose good prefixes are precisely the words in the language of an ERE of size $O(n)$ and whose monitoring (synchronous or asynchronous) requires space $\Omega(2^{2^{\cdot_2 \sqrt[3]{n}}})$ with $\sqrt[3]{n}$ nested power operations.*

7.2 Generating Optimal Monitors for ERE

incorporate RTA '03 paper with Mahesh

Abstract: Software engineers and programmers can easily understand regular patterns, as shown by the immense interest in and the success of scripting languages like Perl, based essentially on regular expression pattern matching. We believe that regular expressions provide an elegant and powerful specification language also for monitoring requirements, because an execution trace of a program is in fact a string of states. Extended regular expressions (EREs) add complementation to regular expressions, which brings additional benefits by allowing one to specify patterns that must not occur during an execution. Complementation gives one the power to express patterns on strings more compactly. In this paper we present a technique to generate optimal monitors from EREs. Our monitors are deterministic finite automata (DFA) and our novel contribution is to generate them using a modern coalgebraic technique called coinduction. Based on experiments with our implementation, which can be publicly tested and used over the web, we believe that our technique is more efficient than the simplistic method based on complementation of automata which can quickly lead to a highly-exponential state explosion.

7.2.1 Introduction

Regular expressions can express patterns in strings in a compact way. They proved very useful in practice; many programming/scripting languages like Perl, Python, Tcl/Tk support regular expressions as core features. Because of their power to express a rich class of patterns, regular expressions, are used not only in computer science but also in various other fields, such as molecular biology [117]. All these applications boast of very efficient implementation of regular expression pattern matching and/or membership algorithms. Moreover, it has been found that compactness of regular expressions can be increased non-elementarily by adding complementation ($\neg R$) to the usual union ($R_1 + R_2$), concatenation ($R_1 \cdot R_2$), and repetition (R^*) operators of regular expressions. These are known as *extended regular expressions* (EREs) and they proved very intuitive and succinct in expressing regular patterns.

Recent trends have shown that the software analysis community is inclining towards scalable techniques for software verification. Works in [86] merged temporal logics with testing, hereby getting the benefits of

both worlds. The Temporal Rover tool (TR) and its follower DB Rover [57] are already commercial. In these tools the Java code is instrumented automatically so that it can check the satisfaction of temporal logic properties at runtime. The MaC tool [115, 127] has been developed to monitor safety properties in interval past time temporal logics. In [141, 144], various algorithms to generate testing automata from temporal logic formulae, are described. Java PathExplorer [83] is a runtime verification environment currently under development at NASA Ames. The Java MultiPathExplorer tool [162] proposes a technique to monitor all equivalent traces that can be extracted from a given execution, thus increasing the coverage of monitoring. [71, 85] present efficient algorithms for monitoring future time temporal logic formulae, while [88] gives a technique to synthesize efficient monitors from past time temporal formulae. [147] uses rewriting to perform runtime monitoring of EREs.

An interesting aspect of EREs is that they can express safety properties compactly, like those encountered in testing and monitoring. By generating automata from logical formulae, several of the works above show that the safety properties expressed by different variants of temporal logics are subclasses of regular languages. The converse is *not* true, because there are regular patterns which cannot be expressed using temporal logics, most notoriously those related to counting; e.g., the regular expression $(0 \cdot (0+1))^*$ saying that every other letter is 0 does not admit an equivalent temporal logic formula. Additionally, EREs tend to be often very natural and intuitive in expressing requirements. For example, let us try to capture the safety property “it should not be the case that in any trace of a traffic light we see green and then immediately red at any point”. The natural and intuitive way to express it in ERE is $\neg((-\emptyset) \cdot \text{green} \cdot \text{red} \cdot (-\emptyset))$, where \emptyset is the empty ERE (no words), so $-\emptyset$ means “anything”.

Previous approaches to ERE membership testing [99, 139, 177, 122, 109] have focussed on developing techniques that are polynomial in both the size of the word and the size of the formulae. The best known result in these approaches is described in [122] where they can check if a word satisfies an ERE in time $O(m \cdot n^2)$ and space $O(m \cdot m + k \cdot n^2)$, where m is the size of the ERE, n is the length of the word, and k is the number of negation/intersection operators. These algorithms, unfortunately, cannot be used for the purpose of monitoring. This is because they are not incremental. They assume the entire word is available before their execution. Additionally, their running time and space requirements are quadratic in the size of the trace. This

is unacceptable when one has a long trace of events and wants to monitor a small ERE, as it is typically the case. This problem is removed in [147] where traces are checked against EREs through incremental rewriting. At present, we do not know if the technique in [147] is optimal or not.

A simple, straightforward, and practical approach is to generate optimal *deterministic finite automata* (DFA) from EREs [103]. This process involves the conversion of each negative sub-component of the ERE to a non-deterministic finite automaton (NFA), determinization of the NFA into a DFA, complementation of the DFA, and then its minimization. The algorithm runs in a bottom-up fashion starting from the innermost negative ERE sub components. This method, although generates the minimal automata, is too complex and cumbersome in practice. Its space requirements can be non-elementarily larger than the initial regular ERE, because negation involves an NFA-to-DFA translation, which implies an exponential blow-up; since negations can be nested, the size of such NFAs or DFAs could be highly exponential.

Our approach is to generate the minimal DFA from an ERE using coinductive techniques. In this paper, the DFA thus generated is called the *optimal monitor* for the given ERE. Currently, we are not aware of any other algorithm that does this conversion in a straightforward way. The complexity of our algorithm seems to be hard to evaluate, because it depends on the size of the minimal DFA associated to an ERE and we are not aware of any lower bound results in this direction. However, experiments are very encouraging. Our implementation, which is available for evaluation on the internet via a CGI server reachable from <http://fs1.cs.uiuc.edu/rv/>, rarely took longer than one second to generate a DFA, and it took only 18 minutes to generate the minimal 107 state DFA for the ERE in Example 16 which was used to show the exponential space lower bound of ERE monitoring in [147].

In a nutshell, in our approach we use the concept of derivatives of an ERE, as described in Subsection 7.2.2. For a given ERE one generates all possible derivatives of the ERE for all possible sequences of events. The size of this set of derivatives depends upon the size of the initial ERE. However, several of these derivative EREs can be equivalent to each other. One can check the equivalence of EREs using coinductive technique as described in Section 7.2.3, that generates a set of equivalent EREs, called *circularities*. In Section 7.2.5, we show how circularities can be used to construct an efficient algorithm that generates optimal DFAs from EREs. In Section 11.4.3, we describe an implementation of this algorithm and give performance analysis

results. We also made available on the internet a CGI interface to this algorithm.

7.2.2 Extended Regular Expressions and Derivatives

In this section we recall extended regular expressions and their derivatives.

Extended Regular Expressions

Extended regular expressions (ERE) define languages by inductively applying union (+), concatenation (\cdot), Kleene Closure ($*$), intersection (\cap), and complementation (\neg). More precisely, for an alphabet E , whose elements are called *events* in this paper, an ERE over E is defined as follows, where $a \in E$:

$$R ::= \emptyset \mid \epsilon \mid a \mid R + R \mid R \cdot R \mid R^* \mid R \cap R \mid \neg R.$$

The language defined by an expression R , denoted by $\mathcal{L}(R)$, is defined inductively as

$$\begin{aligned} \mathcal{L}(\emptyset) &= \emptyset, \\ \mathcal{L}(\epsilon) &= \{\epsilon\}, \\ \mathcal{L}(A) &= \{A\}, \\ \mathcal{L}(R_1 + R_2) &= \mathcal{L}(R_1) \cup \mathcal{L}(R_2), \\ \mathcal{L}(R_1 \cdot R_2) &= \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}(R_1) \text{ and } w_2 \in \mathcal{L}(R_2)\}, \\ \mathcal{L}(R^*) &= (\mathcal{L}(R))^*, \\ \mathcal{L}(R_1 \cap R_2) &= \mathcal{L}(R_1) \cap \mathcal{L}(R_2), \\ \mathcal{L}(\neg R) &= \Sigma^* \setminus \mathcal{L}(R). \end{aligned}$$

Given an ERE, as defined above using union, concatenation, Kleene Closure, intersection and complementation, one can translate it into an equivalent expression that does not have any intersection operation, by applying De Morgan's Law: $R_1 \cap R_2 = \neg(\neg R_1 + \neg R_2)$. The translation only results in a linear blowup in size. Therefore, in the rest of the paper we do not consider expressions containing intersection. More precisely, we only consider EREs of the form

$$R ::= R + R \mid R \cdot R \mid R^* \mid \neg R \mid a \mid \epsilon \mid \emptyset.$$

Derivatives

In this subsection we recall the notion of *derivative*, or “residual” (see [10, 9], where several interesting properties of derivatives are also presented). It is based on the idea of “event consumption”, in the sense that an extended regular expression R and an event a produce another extended regular expression, denoted $R\{a\}$, with the property that for any trace w , $aw \in R$ if and only if $w \in R\{a\}$.

In the rest of the paper assume defined the typical operators on EREs and consider that the operator $_+ _$ is associative and commutative and that the operator $_ \cdot _$ is associative. In other words, reasoning is performed modulo the equations:

$$\begin{aligned} (R_1 + R_2) + R_3 &= R_1 + (R_2 + R_3), \\ R_1 + R_2 &= R_2 + R_1, \\ (R_1 \cdot R_2) \cdot R_3 &= R_1 \cdot (R_2 \cdot R_3). \end{aligned}$$

We next consider an operation $_ \{ _ \}$ which takes an ERE and an event, and give several equations which define its operational semantics recursively, on the structure of regular expressions:

$$\begin{aligned} (R_1 + R_2)\{a\} &= R_1\{a\} + R_2\{a\} & (1) \\ (R_1 \cdot R_2)\{a\} &= (R_1\{a\}) \cdot R_2 + \text{if } (\epsilon \in R_1) \text{ then } R_2\{a\} \text{ else } \emptyset \text{ fi} & (2) \\ (R^*)\{a\} &= (R\{a\}) \cdot R^* & (3) \\ (\neg R)\{a\} &= \neg(R\{a\}) & (4) \\ b\{a\} &= \text{if } (b == a) \text{ then } \epsilon \text{ else } \emptyset \text{ fi} & (5) \\ \epsilon\{a\} &= \emptyset & (6) \\ \emptyset\{a\} &= \emptyset & (7) \end{aligned}$$

The right-hand sides of these equations use operations which we describe next. “if (–) then – else – fi” takes a boolean term and two EREs as arguments and has the expected meaning defined by two equations:

$$\begin{aligned} \text{if } (true) \text{ then } R_1 \text{ else } R_2 \text{ fi} &= R_1 & (8) \\ \text{if } (false) \text{ then } R_1 \text{ else } R_2 \text{ fi} &= R_2 & (9) \end{aligned}$$

We assume a set of equations that properly define boolean expressions and reasoning. Boolean expressions include the constants *true* and *false*, as well as the usual connectors $_ \wedge _$, $_ \vee _$, and *not*. Testing for empty trace membership (which is used by (2)) can be defined via the following equations:

$$\epsilon \in (R_1 + R_2) = (\epsilon \in R_1) \vee (\epsilon \in R_2) \quad (10)$$

$$\epsilon \in (R_1 \cdot R_2) = (\epsilon \in R_1) \wedge (\epsilon \in R_2) \quad (11)$$

$$\epsilon \in (R^*) = \text{true} \quad (12)$$

$$\epsilon \in (\neg R) = \text{not}(\epsilon \in R) \quad (13)$$

$$\epsilon \in b = \text{false} \quad (14)$$

$$\epsilon \in \epsilon = \text{true} \quad (15)$$

$$\epsilon \in \emptyset = \text{false} \quad (16)$$

The 16 equations above are natural and intuitive. [147] shows that these equations, when regarded as rewriting rules are terminating and ground Church-Rosser (modulo associativity and commutativity of $_+ _$ and modulo associativity of $_\cdot _$), so they can be used as a functional procedure to calculate derivatives. Due to the fact that the 16 equations defining the derivatives can generate useless terms, in order to keep EREs compact we also propose defining several *simplifying equations*, including at least the following:

$$\emptyset + R = R,$$

$$\emptyset \cdot R = \emptyset,$$

$$\epsilon \cdot R = R,$$

$$R + R = R.$$

The following result (see, e.g., [147] for a proof) gives a simple procedure, based on derivatives, to test whether a word belongs to the language of an ERE:

Theorem 10 *For any ERE \mathcal{R} and any events a, a_1, a_2, \dots, a_n in A , the following hold:*

1. $a_1 a_2 \dots a_n \in \mathcal{L}(R\{a\})$ if and only if $aa_1 a_2 \dots a_n \in \mathcal{L}(R)$; and
2. $a_1 a_2 \dots a_n \in \mathcal{L}(R)$ if and only if $\epsilon \in R\{a_1\}\{a_2\}\dots\{a_n\}$.

7.2.3 Hidden Logic and Coinduction

We use circular coinduction, defined rigorously in the context of hidden logics and implemented in the BOBJ system [145, 73, 74], to test whether two EREs are equivalent, that is, if they have the same language. Since the goal of this paper is to translate an ERE into a minimal DFA, standard techniques for checking equivalence, such as translating the two expressions into DFAs and then comparing those, do not make sense in this framework. A particularly appealing aspect of circular coinduction in the framework of

EREs is that it does not only show that two EREs are equivalent, but also generates a larger set of equivalent EREs which will all be used in order to generate the target DFA.

Hidden logic is a natural extension of algebraic specification which benefits of a series of generalizations in order to capture various natural notions of behavioral equivalence found in the literature. It distinguishes *visible* sorts for data from *hidden* sorts for states, with states *behaviorally equivalent* if and only if they are indistinguishable under a formally given set of experiments. To keep the presentation simple and self contained, in this section we define an oversimplified version of hidden logic together with its associated circular coinduction proof rule, still general enough to support defining and proving EREs behaviorally equivalent.

Algebraic Preliminaries

The reader is assumed familiar with basic equational logic and algebra in this section. We recall a few notions in order to just make our notational conventions precise. An S -sorted signature Σ is a set of sorts/types S together with operational symbols on those, and a Σ -algebra A is a collection of sets $\{A_s \mid s \in S\}$ and a collection of functions appropriately defined on those sets, one for each operational symbol. Given an S -sorted signature Σ and an S -indexed set of variables Z , let $T_\Sigma(Z)$ denote the Σ -term algebra over variables in Z . If $V \subseteq S$ then $\Sigma|_V$ is a V -sorted signature consisting of all those operations in Σ with sorts entirely in V . We may let $\sigma(X)$ denote the term $\sigma(x_1, \dots, x_n)$ when the number of arguments of σ and their order and sorts are not important. If only one argument is important, then to simplify writing we place it at the beginning; for example, $\sigma(t, X)$ is a term having σ as root with only variables as arguments except one, and we do not care which one, which is t . If t is a Σ -term of sort s' over a special variable $*$ of sort s and A is a Σ -algebra, then $A_t : A_s \rightarrow A_{s'}$ is the usual interpretation of t in A .

7.2.4 Behavioral Equivalence, Satisfaction and Specification

Given disjoint sets V, H called *visible* and *hidden sorts*, a *hidden* (V, H) -signature, say Σ , is a many sorted $(V \cup H)$ -signature. A *hidden subsignature* of Σ is a hidden (V, H) -signature Γ with $\Gamma \subseteq \Sigma$ and $\Gamma|_V = \Sigma|_V$. The *data signature* is $\Sigma|_V$. An operation of visible result not in $\Sigma|_V$ is called an *attribute*, and a hidden sorted operation is called a *method*.

Unless otherwise stated, the rest of this section assumes fixed a hidden signature Σ with a fixed subsignature Γ . Informally, Σ -algebras are universes

of possible states of a system, i.e., “black boxes,” where one is only concerned with behavior under experiments with operations in Γ , where an experiment is an observation of a system attribute after perturbation; this is formalized below.

A Γ -context for sort $s \in V \cup H$ is a term in $T_\Gamma(\{ * : s \})$ with one occurrence of $*$. A Γ -context of visible result sort is called a Γ -experiment. If c is a context for sort h and $t \in T_{\Sigma, h}$ then $c[t]$ denotes the term obtained from c by substituting t for $*$; we may also write $c[*]$ for the context itself.

Given a hidden Σ -algebra A with a hidden subsignature Γ , for sorts $s \in (V \cup H)$, we define Γ -behavioral equivalence of $a, a' \in A_s$ by $a \equiv_\Sigma^\Gamma a'$ iff $A_c(a) = A_c(a')$ for all Γ -experiments c ; we may write \equiv instead of \equiv_Σ^Γ when Σ and Γ can be inferred from context. We require that all operations in Σ are compatible with \equiv_Σ^Γ . Note that behavioral equivalence is the identity on visible sorts, since the trivial contexts $* : v$ are experiments for all $v \in V$. A major result in hidden logics, underlying the foundations of coinduction, is that Γ -behavioral equivalence is the largest equivalence which is identity on visible sorts and which is compatible with the operations in Γ .

Behavioral satisfaction of equations can now be naturally defined in terms of behavioral equivalence. A hidden Σ -algebra A Γ -behaviorally satisfies a Σ -equation $(\forall X) t = t'$, say e , iff for each $\theta : X \rightarrow A$, $\theta(t) \equiv_\Sigma^\Gamma \theta(t')$; in this case we write $A \models_\Sigma^\Gamma e$. If E is a set of Σ -equations we then write $A \models_\Sigma^\Gamma E$ when A Γ -behaviorally satisfies each Σ -equation in E . We may omit Σ and/or Γ from \models_Σ^Γ when they are clear.

A behavioral Σ -specification is a triple (Σ, Γ, E) where Σ is a hidden signature, Γ is a hidden subsignature of Σ , and E is a set of Σ -sentences equations. Non-data Γ -operations (i.e., in $\Gamma - \Sigma|_V$) are called *behavioral*. A Σ -algebra A *behaviorally satisfies* a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$ iff $A \models_\Sigma^\Gamma E$, in which case we write $A \models \mathcal{B}$; also $\mathcal{B} \models e$ iff $A \models \mathcal{B}$ implies $A \models_\Sigma^\Gamma e$.

EREs can be very naturally defined as a behavioral specification. The enormous benefit of doing so is that the behavioral inference, including most importantly coinduction, provide a *decision procedure* for equivalence of EREs. [73] shows how standard regular expressions (without negation) can be defined as a behavioral specification, a BOBJ implementation, and also how BOBJ with its circular coinductive rewriting algorithm can prove automatically several equivalences of regular expressions. Related interesting work can also be found in [156]. In this paper we extend that to general EREs, generate minimal observer monitors, and also give several other

examples.

Example 10 *A behavioral specification of EREs defines a set of two visible sorts $V = \{Bool, Event\}$, one hidden sort $H = \{Ere\}$, one behavioral attribute $\epsilon \in _ : Ere \rightarrow Bool$ and one behavioral method, the derivative, $_[-] : Ere \times Event \rightarrow Ere$, together with all the other operations in Subsection 7.2.2 defining EREs, including the events in E which are defined as visible constants of sort $Event$, and all the equations in Subsection 7.2.2. We call it the ERE behavioral specification and let \mathcal{B}_{ERE} denote it.*

*Since the only behavioral operators are the test for ϵ membership and the derivative, it follows that the experiments have exactly the form $\epsilon \in * \{a_1\} \{a_2\} \dots \{a_n\}$, for any events a_1, a_2, \dots, a_n . In other words, an experiment consists of a series of derivations followed by an ϵ membership test, and therefore two regular expressions are behavioral equivalent if and only if they cannot be distinguished by such experiments. Notice that the above reasoning applies within any algebra satisfying the presented behavioral specification. The one we are interested in is, of course, the free one, whose set carriers contain exactly the extended regular expressions as presented in Subsection 7.2.2, and the operations have the obvious interpretations. We informally call it the ERE algebra.*

Letting \equiv denote the behavioral equivalence relation generated on the ERE algebra, then Theorem 10 immediately yields the following important result.

Theorem 11 *If R_1 and R_2 are two EREs then $R_1 \equiv R_2$ if and only if $\mathcal{L}(R_1) = \mathcal{L}(R_2)$.*

This theorem allows us to prove equivalence of EREs by making use of behavioral inference in the ERE behavioral specification, from now on simply referred to by \mathcal{B} , including (especially) circular coinduction. The next section shows how circular coinduction works and how it can be used to show EREs equivalent.

Circular Coinduction as an Inference Rule

In the simplified version of hidden logics defined above, the usual equational inference rules, i.e., reflexivity, symmetry, transitivity, substitution and congruence [145] are all sound for behavioral satisfaction. However, equational reasoning can derive only a very limited amount of interesting behavioral equalities. For that reason, *circular coinduction* has been developed

as a very powerful automated technique to show behavioral equivalence. We let \Vdash denote the relation being defined by the equational rules plus circular coinduction, for deduction from a specification to an equation.

Before we present circular coinduction formally, we give the reader some intuitions by duality to structural induction. The reader who is only interested in using the presented procedure or who is not familiar with structural induction, can skip this paragraph. Inductive proofs show equality of terms $t(x), t'(x)$ over a given variable x (seen as a constant) by showing $t(\sigma(x))$ equals $t'(\sigma(x))$ for all σ in a basis, while circular coinduction shows terms t, t' behaviorally equivalent by showing equivalence of $\delta(t)$ and $\delta(t')$ for all behavioral operations δ . Coinduction applies behavioral operations at the top, while structural induction applies generator/constructor operations at the bottom. Both induction and circular coinduction assume some “frozen” instances of t, t' equal when checking the inductive/coinductive step: for induction, the terms are frozen at the bottom by replacing the induction variable by a constant, so that no other terms can be placed beneath the induction variable, while for coinduction, the terms are frozen at the top, so that they cannot be used as subterms of other terms (with some important but subtle exceptions which are not needed here; see [74]).

Freezing terms at the top is elegantly handled by a simple trick. Suppose every specification has a special visible sort b , and for each (hidden or visible) sort s in the specification, a special operation $[-] : s \rightarrow b$. No equations are assumed for these operations and no user defined sentence can refer to them; they are there for technical reasons. Thus, with just the equational inference rules, for any behavioral specification \mathcal{B} and any equation $(\forall X) t = t'$, it is necessarily the case that $\mathcal{B} \Vdash (\forall X) t = t'$ iff $\mathcal{B} \Vdash (\forall X) [t] = [t']$. The rule below preserves this property. Let the sort of t, t' be hidden; then

Circular Coinduction:

$$\frac{\mathcal{B} \cup \{(\forall X) [t] = [t']\} \Vdash (\forall X, W) [\delta(t, W)] = [\delta(t', W)], \text{ for all appropriate } \delta \in \Gamma}{\mathcal{B} \Vdash (\forall X) t = t'}$$

We call the equation $(\forall X) [t] = [t']$ added to \mathcal{B} a **circularity**; it could just as well have been called a coinduction hypothesis or a co-hypothesis, but we find the first name more intuitive because from a coalgebraic point of view, coinduction is all about finding circularities.

Theorem 12 *The usual equational inference rules together with Circular Coinduction are sound. That means that if $\mathcal{B} \Vdash (\forall X) t = t'$ and $\text{sort}(t, t') \neq b$, or if $\mathcal{B} \Vdash (\forall X) [t] = [t']$, then $\mathcal{B} \models (\forall X) t = t'$.*

Example 11 *Suppose that we want to show that the EREs $(a + b)^*$ and $(a^*b^*)^*$ admit the same language. By Theorem 11, we can instead show that $\mathcal{B}_{ERE} \models (\forall \emptyset) (a + b)^* = (a^*b^*)^*$. Notice that a and b are treated as constant events here; one can also prove the result when a and b are variables, but one would need to first make use of the theorem of hidden constants [145]. To simplify writing, we omit the empty quantifier of equations. By the Circular Coinduction rule, one generates the following three proof obligations*

$$\begin{aligned} \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [\epsilon \in (a + b)^*] = [\epsilon \in (a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [(a + b)^*\{a\}] = [(a^*b^*)^*\{a\}], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [(a + b)^*\{b\}] = [(a^*b^*)^*\{b\}]. \end{aligned}$$

The first proof task follows immediately by using the equations in \mathcal{B} as rewriting rules, while the other two tasks reduce to

$$\begin{aligned} \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [(a + b)^*] = [a^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [(a + b)^*] = [b^*(a^*b^*)^*]. \end{aligned}$$

By applying Circular Coinduction twice, after simplifying the two obvious proof tasks stating the ϵ membership, one gets the following four proof obligations

$$\begin{aligned} \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [a^*(a^*b^*)^*]\} &\Vdash [(a + b)^*\{a\}] = [a^*(a^*b^*)^*\{a\}], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [a^*(a^*b^*)^*]\} &\Vdash [(a + b)^*\{b\}] = [a^*(a^*b^*)^*\{b\}], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [b^*(a^*b^*)^*]\} &\Vdash [(a + b)^*\{a\}] = [b^*(a^*b^*)^*\{a\}], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [b^*(a^*b^*)^*]\} &\Vdash [(a + b)^*\{b\}] = [b^*(a^*b^*)^*\{b\}], \end{aligned}$$

which, after simplification translate into

$$\begin{aligned} \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [a^*(a^*b^*)^*]\} &\Vdash [(a + b)^*] = [a^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [a^*(a^*b^*)^*]\} &\Vdash [(a + b)^*] = [b^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [b^*(a^*b^*)^*]\} &\Vdash [(a + b)^*] = [a^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [b^*(a^*b^*)^*]\} &\Vdash [(a + b)^*] = [b^*(a^*b^*)^*], \end{aligned}$$

Again by applying circular coinduction we get

$$\begin{array}{l}
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} \Vdash \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} \Vdash \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} \Vdash \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} \Vdash
\end{array}$$

which now follow all immediately. Notice that BOBJ uses the newly added (to \mathcal{B}_{ERE}) equations as rewriting rules when it applies its circular coinductive rewriting algorithm, so the proof above is done slightly differently, but entirely automatically.

Example 12 Suppose now that one wants to show that $\neg(a^*b) \equiv \epsilon + a^* + (a+b)^*b(a+b)(a+b)^*$. One can also do it entirely automatically by circular coinduction as above, generating the following list of circularities:

$$\begin{array}{lcl}
[\neg(a^*b)] & = & [\epsilon + a^* + (a+b)^*b(a+b)(a+b)^*], \\
[\neg(\epsilon)] & = & [(a+b)^*b(a+b)(a+b)^* + (a+b)(a+b)^*], \\
[\neg(\emptyset)] & = & [(a+b)^*b(a+b)(a+b)^* + (a+b)^*], \\
[\neg(\emptyset)] & = & [(a+b)^*b(a+b)(a+b)^* + (a+b)(a+b)^* + (a+b)^*].
\end{array}$$

Example 13 One can also show by circular coinduction that concrete EREs satisfy systems of guarded equations. This is an interesting but unrelated subject, so we do not discuss it in depth here. However, we show how easily one can prove by coinduction that a^*b is the solution of the equation $R = a \cdot R + b$. This equation can be given by adding a new ERE constant r to \mathcal{B}_{ERE} , together with the equations $\epsilon \in r = \text{false}$, $r\{a\} = r$, and $r\{b\} = \epsilon$. Circular Coinduction applied on the goal $r = a^*b$ generates the proof tasks:

$$\begin{array}{lcl}
\mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} & \Vdash & [\epsilon \in r] = [\epsilon \in a^*b], \\
\mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} & \Vdash & [r\{a\}] = [a^*b\{a\}], \\
\mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} & \Vdash & [r\{b\}] = [a^*b\{b\}],
\end{array}$$

which all follow immediately.

The following says that circular coinduction provides a decision procedure for equivalence of EREs.

Theorem 13 *If R_1 and R_2 are two EREs, then $\mathcal{L}(R_1) = \mathcal{L}(R_2)$ if and only if $\mathcal{B}_{ERE} \Vdash R_1 = R_2$. Moreover, since the rules in \mathcal{B}_{ERE} are ground Church-Rosser and terminating, circular coinductive rewriting [73, 74], which iteratively rewrites proof tasks to their normal forms followed by a one step coinduction if needed, gives a decision procedure for ERE equivalence.*

7.2.5 Generating Minimal DFA Monitors by Coinduction

In this section we show how one can use the set of circularities generated by applying the circular coinduction rules in order to generate a minimal DFA from any ERE. This DFA can then be used as an optimal monitor for that ERE. The main idea here is to associate states in DFA to EREs obtained by deriving the initial ERE; when a new ERE is generated, it is tested for equivalence with all the other already generated EREs by using the coinductive procedure presented in the previous section. A crucial observation which significantly reduces the complexity of our procedure is that, once an equivalence is proved by circular coinductive rewriting, the entire set of circularities accumulated represent equivalent EREs. These can be used to later quickly infer the other equivalences, without having to generate the same circularities over and over again.

Since BOBJ does not (yet) provide any mechanism to return the set of circularities accumulated after proving a given behavioral equivalence, we were unable to use BOBJ to implement our optimal monitor generator. Instead, we have implemented our own version of coinductive rewriting engine for EREs, which is described below.

We are given an initial ERE R_0 over alphabet A and from that we want to generate the equivalent minimal DFA $D = (S, A, \delta, s_0, F)$, where S is the set of states, $\delta : S \times A \rightarrow S$ is the transition function, s_0 is the initial state, and $F \subseteq S$ is the set of final states. The coinductive rewriting engine explicitly accumulates the proven circularities in a set. The set is initialized to an empty set at the beginning of the algorithm. It is updated with the accumulated circularities whenever we prove equivalence of two regular expressions in the algorithm. The algorithm maintains the set of states S in the form of non-equivalent EREs. At the beginning of the algorithm S is initialized with a single element, which is the given ERE R_0 . Next, we generate all the derivatives of the initial ERE one by one in a depth first manner. A derivative $R_x = R\{x\}$ is added to the set S , if the set does not contain any ERE equivalent to the derivative R_x . We then extend the transition function by setting $\delta(R, x) = R_x$. If an ERE R' equivalent to the

derivative already exists in the set S , we extend the transition function by setting $\delta(R, x) = R'$. To check if an ERE equivalent to the derivative R_x already exists in the set S , we sequentially go through all the elements of the set S and try to prove its equivalence with R_x . In testing the equivalence we first add the set of circularities to the initial \mathcal{B} . Then we invoke the coinductive procedure. If for some ERE $R' \in S$, we are able to prove that $R' \equiv R_x$ i.e $\mathcal{B} \cup Eq_{\text{all}} \cup Eq_{\text{new}} \Vdash R' = R_x$, then we add the new equivalences Eq_{new} , created by the coinductive procedure, to the set of circularities. Thus we reuse the already proven equivalences in future proofs.

The derivatives of the initial ERE R_0 with respect to all events in the alphabet A are generated in a depth first fashion. The pseudo code for the whole algorithm is given in Figure 1.

```

dfs( $R$ )
begin
  foreach  $x \in A$  do
     $R_x \leftarrow R\{x\}$ ;
    if  $\exists R' \in S$  such that  $\mathcal{B} \cup Eq_{\text{all}} \cup Eq_{\text{new}} \Vdash R' = R_x$  then
       $\delta(R, x) = R'$ ;  $Eq_{\text{all}} \leftarrow Eq_{\text{all}} \cup Eq_{\text{new}}$ 
    else  $S \leftarrow S \cup \{R_x\}$ ;  $\delta(R, x) = R_x$ ; dfs( $R_x$ ); fi
  endfor
end

```

Figure 7.1: ERE to minimal DFA generation algorithm

In the procedure **dfs** the set of final states F consists of the EREs from S which contain ϵ . This can be tested efficiently using the equations (10-16) in Subsection 7.2.2. The DFA generated by the procedure **dfs** may now contain some states which are non-final and from which the DFA can never reach a final state. We remove these redundant states by doing a breadth first search in backward direction from the final states. This can be done in time linear in the size of the DFA.

Theorem 14 *If D is the DFA generated for a given ERE R by the above algorithm then*

1. $\mathcal{L}(D) = \mathcal{L}(R)$,
2. D is the minimal DFA accepting $\mathcal{L}(R)$.

Proof: Suppose $a_1a_2 \dots a_n \in \mathcal{L}(R)$. Then $\epsilon \in R\{a_1\}\{a_2\} \dots \{a_n\}$. If $R_i = R\{a_1\}\{a_2\} \dots \{a_i\}$ then $R_{i+1} = R_i\{a_{i+1}\}$. To prove that $a_1a_2 \dots a_n \in \mathcal{L}(D)$, we use induction to show that for each $1 \leq i \leq n$, $R_i \equiv \delta(R, a_1a_2 \dots a_i)$. For the base case if $R_1 \equiv R\{a_1\}$ then **dfs** extends the transition function by setting $\delta(R, a_1) = R$. Therefore, $R_1 \equiv R = \delta(R, a_1)$. If $R_1 \not\equiv R$ then **dfs** extends δ by setting $\delta(R, a_1) = R_1$. So $R_1 \equiv \delta(R, a_1)$ holds in this case also. For the induction step let us assume that $R_i \equiv R' = \delta(R, a_1a_2 \dots a_i)$. If $\delta(R', a_{i+1}) = R''$ then from the **dfs** procedure we can see that $R'' \equiv R'\{a_{i+1}\}$. However, $R_i\{a_{i+1}\} \equiv R'\{a_{i+1}\}$. So $R_{i+1} \equiv R'' = \delta(R', a_{i+1}) = \delta(R, a_1a_2 \dots a_{i+1})$. Also notice $\epsilon \in R_n \equiv \delta(R, a_1a_2 \dots a_n)$; this implies that $\delta(R, a_1a_2 \dots a_n)$ is a final state and hence $a_1a_2 \dots a_n \in \mathcal{L}(D)$.

Now suppose $a_1a_2 \dots a_n \in \mathcal{L}(D)$. The proof that $a_1a_2 \dots a_n \in \mathcal{L}(R)$ goes in a similar way by showing that $R_i \equiv \delta(R, a_1, a_2 \dots a_i)$. \square

7.2.6 Implementation and Evaluation

We have implemented the coinductive rewriting engine in the rewriting specification language Maude 2.0 [43]. The interested readers can download the implementation from the website <http://fsl.cs.uiuc.edu/rv/>. The operations on extended regular languages that are supported by our implementation are \sim for negation, $*$ for Kleene Closure, $_$ for concatenation, $\&$ for intersection, and $+$ for union in increasing order of precedence. Here, the intersection operator $\&$ is a syntactic sugar and is translated to an ERE containing union and negation using De Morgan's Law:

$$\text{eq } R1 \ \& \ R2 = \sim (\sim R1 + \sim R2) \ .$$

To evaluate the performance of the algorithm we have generated the minimal DFA for all possible EREs of size up to 9. Surprisingly, the size of any DFA for EREs of size up to 9 did not exceed 9. Here the number of states gives the size of a DFA. The following table shows the performance of our procedure for the worst EREs of a given size. The code is executed on a

Pentium 4 2.4GHz, 4 GB RAM linux machine.

Size	ERE	no. of states in DFA	Time (ms)	Rewrites
4	$\neg (a \ b)$	4	< 1	863
5	$(a \neg b)^*$	4	< 1	1370
6	$\neg ((a \neg b)^*)$	4	1	1453
7	$\neg (a \neg a \ a)$	6	1	2261
8	$\neg ((a \neg b)^* \ b)$	7	1	3778
9	$\neg (a \neg a \ b) \ b$	9	5	9717

Example 14 In particular, for the ERE $\neg (a \neg a \ b) \ b$ the generated minimal DFA is given in Figure 7.2.

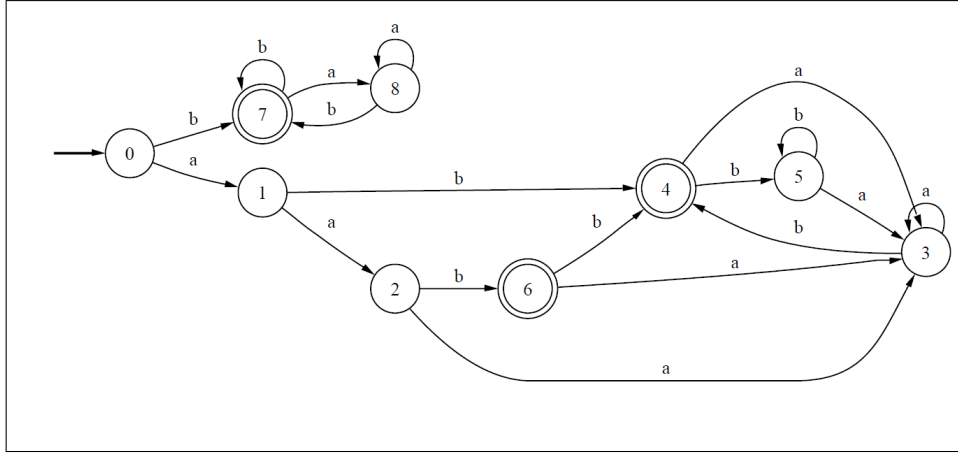


Figure 7.2: $\neg (a \neg a \ b) \ b$

Example 15 The ERE $\neg ((\neg \text{empty}) (\text{green red}) (\neg \text{empty}))$ states the safety property that it should not be the case that in any trace of a traffic light we see green and red consecutively at any point. The set of events are assumed to be $\{\text{green}, \text{red}, \text{yellow}\}$. We think that this is the most intuitive and natural expression for this safety property. The implementation took 1ms and 1663 rewrites to generate the minimal DFA with 2 states. The DFA is given in Figure 7.3.

However for large EREs the algorithm may take a long time to generate a minimal DFA. The size of the generated DFA may grow non-elementarily

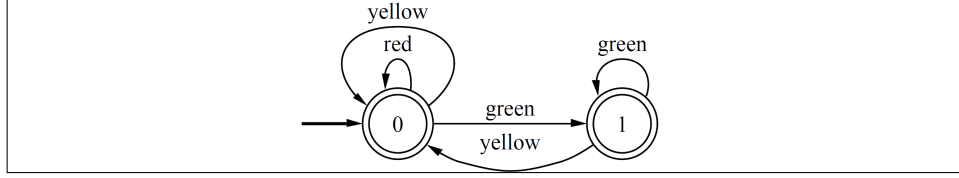


Figure 7.3: $\neg ((\neg \text{empty}) (\text{green red}) (\neg \text{empty}))$

in the worst case. We generated DFAs for some complex EREs of larger sizes and got relatively promising results. One such sample result is as follows.

Example 16 *Let us consider the following ERE of size 110*

$$\begin{aligned}
 &(\neg \$)^* \$ (\neg \$)^* \cap \\
 & (0 + 1 + \#)^* \# (\\
 & ((0 + 1)0\#(0 + 1 + \#)^* \$ (0 + 1)0 + (0 + 1)1\#(0 + 1 + \#)^* \$ (0 + 1)1) \\
 & \cap (0(0 + 1)\#(0 + 1 + \#)^* \$ 0(0 + 1) + 1(0 + 1)\#(0 + 1 + \#)^* \$ 1(0 + 1))).
 \end{aligned}$$

This ERE accepts the language L_2 , where

$$L_k = \{\sigma \# w \# \sigma' \$ w \mid w \in \{0, 1\}^k \text{ and } \sigma, \sigma' \in \{0, 1, \#\}^*\}$$

The language L_k was first introduced in [31] to show the power of alternation, used in [147] to show an exponential lower bound on ERE monitoring, and in [119, 120] to show the lower bounds for model checking. Our implementation took almost 18 minutes to generate the minimal DFA of size 107 and in the process it performed 1,374,089,220 rewrites.

The above example shows that the procedure can take a large amount of time and space to generate DFAs for large EREs. To avoid the computation associated with the generation of minimal DFA we plan to maintain a database of EREs and their corresponding minimal DFAs on the internet. Whenever someone wants to generate the minimal DFA for a given ERE he/she can look up the internet database for the minimal DFA. If the ERE and the corresponding DFA exists in the database he/she can retrieve the corresponding DFA and use it as a monitor. Otherwise, he/she can generate the minimal DFA for the ERE and submit it to the internet database to create a new entry. The database will check the equivalence of the submitted ERE and the corresponding minimal DFA and insert it in the database. In this way one can avoid the computation of generating minimal DFA if it is already done by someone else. To further reduce the computation, circularities could also be stored in the database.

Online Monitor Generation and Visualization

We have extended our implementation to create an internet server for optimal monitor generation that can be accessed from the url <http://fsl.cs.uiuc.edu/rv/>. Given an ERE the server generates the optimal DFA monitor for a user. The user submits the ERE through a web based form. A CGI script handling the web form takes the submitted ERE as an input, invokes the Maude implementation to generate the minimal DFA, and presents it to the user either as a graphical or a textual representation. To generate the graphical representation of the DFA we are currently using the GraphViz tool [66].

We presented a new technique to generate optimal monitors for extended regular expressions, which avoids the traditional technique based on complementation of automata, that we think is quite complex and not necessary. Instead, we have considered the (co)algebraic definition of EREs and applied coinductive inferencing techniques in an innovative way to generate the minimal DFA. Our approach to store already proven equivalences has resulted into a very efficient and straightforward algorithm to generate minimal DFA. We have evaluated our implementation on several hundreds EREs and have got promising results in terms of running time. Finally we have installed a server on the internet which can generate the optimal DFA for a given ERE.

At least two major contributions have been made. Firstly, we have shown that coinduction is a viable and quite practical method to prove equivalence of extended regular expressions. Previously this was done only for regular expressions without complementation. Secondly, building on the coinductive technique, we have devised an algorithm to generate minimal DFAs from EREs. At present we have no bound for the size of the optimal DFA, but we know for sure that the DFAs we generate are indeed optimal. However we know that the size of an optimal DFA is bounded by some exponential in the size of the ERE. As future work, it seems interesting to investigate the size of minimal DFAs generated from EREs, and also to apply our coinductive techniques to generate monitors for other logics, such as temporal logics.

Exercises

Exercise 10 *Does the lower bound result in Section 7.1 hold for semi-extended regular expressions, that is, for regular expressions extended only with intersection (\cap) but not negation (\neg)? If yes, then sketch a proof. If no, then explain why not and try to find another lower bound.*

Exercise 11 *Implement the derivative-based ERE membership checking algorithm described in Section 7.2.2 (see also Theorem 10). Use your favorite programming language.*

Exercise* 12 *Implement the coinductive monitor synthesis for ERE described in Section 7.2.5.*

Chapter 8

Monitor Synthesis: Linear Temporal Logic (LTL)

add co-inductive algorithm

8.1 Finite Trace Future Time Linear Temporal Logic

Classical (infinite trace) linear temporal logic (LTL) [143, 128, 130] provides in addition to the propositional logic operators the temporal operators \Box (always), \Diamond (eventually), \cup (until), and \circ (next). An LTL standard model is a function $t : \mathcal{N}^+ \rightarrow 2^{\mathcal{P}}$ for some set of atomic propositions \mathcal{P} , i.e., an infinite trace over the alphabet $2^{\mathcal{P}}$, which maps each time point (a natural number) into the set of propositions that hold at that point. The operators have the following interpretation on such an infinite trace. Assume formulae x and y . The formula $\Box x$ holds if x holds in all time points, while $\Diamond x$ holds if x holds in some future time point. The formula $x \cup y$ (x until y) holds if y holds in some future time point, and until then x holds (so we consider strict until). Finally, $\circ x$ holds for a trace if x holds in the suffix trace starting in the next (the second) time point. The propositional operators have their obvious meaning. For example, the formula $\Box (x \rightarrow \Diamond y)$ is true if for any time point (\Box) it holds that if x is true then eventually (\Diamond) y is true. It is standard to define a core LTL using only atomic propositions, the propositional operators \neg (not) and \wedge (and), and the temporal operators \circ and \cup , and then define all other propositional and temporal operators as derived constructs. Standard equations are $\Diamond x = \text{true} \cup x$ and $\Box x =$

!<>!X.

Our goal is to develop a framework for testing software systems using temporal logic. Tests are performed on finite execution traces and we therefore need to formalize what it means for a *finite trace* to satisfy an LTL formula. We first present a semantics of finite trace LTL using standard mathematical notation. Then we present a specification in Maude of a finite trace semantics. Whereas the former semantics uses universal and existential quantification, the second Maude specification is defined using recursive definitions that have a straightforward operational rewriting interpretation and which therefore can be executed.

8.1.1 Finite Trace Semantics

As mentioned in Subsection 2.1.2, a trace is viewed as a non-empty finite sequence of program states, each state denoting the set of propositions that hold at that state. We shall first outline the finite trace LTL semantics using standard mathematical notation rather than Maude notation. The debatable issue here is what happens at the end of the trace. The choice to validate or invalidate all the atomic propositions does not work in practice, because there might be propositions whose values are always opposite to each other, such as, for example, “gate up” and “gate down”. Driven by experiments, we found that a more reasonable assumption is to regard a finite trace as an infinite stationary trace in which the last event is repeated infinitely.

Assume two total functions on traces, $head : \text{Trace} \rightarrow \text{Event}$ returning the head event of a trace and $length$ returning the length of a finite trace, and a partial function $tail : \text{Trace} \rightarrow \text{Trace}$ for taking the tail of a trace. That is, $head(e, t) = head(e) = e$, $tail(e, t) = t$, and $length(e) = 1$ and $length(e, t) = 1 + length(t)$. Assume further for any trace t , that t_i denotes the suffix trace that starts at position i , with positions starting at 1. The satisfaction relation $\models \subseteq \text{Trace} \times \text{Formula}$ defines when a trace t satisfies a formula f , written $t \models f$, and is defined inductively over the structure of the formulae as follows, where A is any atomic proposition and X and Y are any formulae:

$t \models \text{true}$	iff	true ,
$t \models \text{false}$	iff	false ,
$t \models A$	iff	$A \in \text{head}(t)$,
$t \models X \wedge Y$	iff	$t \models X$ and $t \models Y$,
$t \models X \vee Y$	iff	$t \models X$ or $t \models Y$,
$t \models \circ X$	iff	(if $\text{tail}(t)$ is defined then $\text{tail}(t) \models X$ else $t \models X$),
$t \models \langle \rangle X$	iff	$(\exists i \leq \text{length}(t)) t_i \models X$,
$t \models \Box X$	iff	$(\forall i \leq \text{length}(t)) t_i \models X$,
$t \models X \cup Y$	iff	$(\exists i \leq \text{length}(t)) (t_i \models Y \text{ and } (\forall j < i) t_j \models X)$.

The semantics of the “next” operator reflects perhaps best the stationarity assumption of last events in finite traces.

Notice that finite trace LTL can behave quite differently from standard infinite trace LTL. For example, there are formulae which are not valid in infinite trace LTL but valid in finite trace LTL, such as $\langle \rangle (\Box A \vee \Box !A)$ for any atomic proposition A , and there are formulae which are satisfiable in infinite trace LTL and not satisfiable in finite trace LTL, such as the negation of the above. The formula above is satisfied by any finite trace because the last event/state in the trace either contains A or it does not.

8.1.2 Finite Trace Semantics in Maude

Now it can be relatively easily seen that the following Maude specification correctly “defines” the finite trace semantics of LTL described above. The only important deviation from the rigorous mathematical formulation described above is that the quantifiers over finite sets of indexes are expressed recursively.

```
fmod LTL is
  extending PROP-CALC .
*** syntax
  op []_ : Formula -> Formula [prec 11] .
  op <>_ : Formula -> Formula [prec 11] .
  op _U_ : Formula Formula -> Formula [prec 14] .
  op o_ : Formula -> Formula [prec 11] .
*** semantics
  vars X Y : Formula .
  var E : Event .
  var T : Trace .
  eq E |= o X = E |= X .
  eq E,T |= o X = T |= X .
```

```

eq E    |= <> X  = E |= X .
eq E,T  |= <> X  = E,T |= X or T |= <> X .
eq E    |= [] X  = E |= X .
eq E,T  |= [] X  = E,T |= X and T |= [] X .
eq E    |= X U Y = E |= Y .
eq E,T  |= X U Y = E,T |= Y or E,T |= X and T |= X U Y .
endfm

```

Notice that only the temporal operators needed declarations and semantics, the others being already defined in PROP-CALC and LOGICS-BASIC, and that the definitions that involved the functions *head* and *tail* were replaced by two alternative equations.

One can now directly verify LTL properties on finite traces using Maude's rewriting engine. Consider as an example a traffic light that switches between the colors *green*, *yellow*, and *red*. The LTL property that after *green* comes *yellow*, and its negation, can now be verified on a finite trace using Maude's rewriting engine, by typing verification commands to Maude such as:

```

reduce green, yellow, red, green, yellow, red, green, yellow, red, red
      |= [](green -> !red U yellow) .
reduce green, yellow, red, green, yellow, red, green, yellow, red, red
      |= !([](green -> !red U yellow)) .

```

which should return the expected answers, i.e., **true** and **false**, respectively. The algorithm above does nothing but blindly follows the mathematical definition of satisfaction and even runs reasonably fast for relatively small traces. For example, it takes¹ about 30ms (74k rewrite steps) to reduce the first formula above and less than 1s (254k rewrite steps) to reduce the second on traces of 100 events (10 times larger than the above). Unfortunately, this algorithm does not seem to be tractable for large event traces, even if run on very fast platforms. As a concrete practical example, it took Maude 7.3 million rewriting steps (3 seconds) to reduce the first formula above and 2.4 billion steps (1000 seconds) for the second on traces of 1,000 events; it could not finish in one night (more than 10 hours) the reduction of the second formula on a trace of 10,000 events. Since the event traces generated by an executing program can easily be larger than 10,000 events, the trivial algorithm above cannot be used in practice.

A rigorous complexity analysis of the algorithm above is hard (because it has to take into consideration the evaluation strategy used by Maude for

¹On a 1.7GHz, 1Gb memory PC.

terms of sort `Bool`) and not worth the effort. However, a simplified worse-case analysis can be easily made if one only counts the maximum number of atoms of the form `event` \models `atom` that can occur during the rewriting of a satisfaction term, as if all the Boolean reductions were applied after all the other reductions: let us consider a formula $X = \Box(\Box(\dots(\Box A)\dots))$ where the always operator is nested m times, and a trace T of size n , and let $T(n, m)$ be the total number of basic satisfactions `event` \models `atom` that occur in the normal form of the term $T \models X$ if no Boolean reductions were applied. Then, the recurrence formula $T(n, m) = T(n - 1, m) + T(n, m - 1)$ follows immediately from the specification above. Since $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$, it follows that $T(n, m) > \binom{n}{m}$, that is, $T(n, m) = \Omega(n^m)$, which is of course unacceptable.

8.2 A Backwards, Asynchronous, but Efficient Algorithm

The satisfaction relation above for finite trace LTL can hence be defined recursively, both on the structure of the formulae and on the size of the execution trace. As is often the case for functions defined this way, an efficient *dynamic programming* algorithm can be generated from any LTL formula. We first show how such an algorithm looks for a particular formula, and then present the main algorithm generator. The work in this section appeared as a technical report [149], but for a slightly different finite trace LTL, namely one in which all the atomic propositions were considered *false* at the end of the trace. As explained previously in the paper, we are now in the favor of a semantics where traces are considered stationary in their last event. The generated dynamic programming algorithms are as efficient as they can be and one can hope: linear in both the trace and the LTL formula. Unfortunately, they need to traverse the execution trace backwards, so they are trace storing and asynchronous. However, a similar but dual technique applies to past time LTL, producing very efficient forwards and synchronous algorithms [88, 87].

8.2.1 An Example

The formula we choose below is artificial (and will not be used later in the paper), but contains all four temporal operators. We believe that this example would practically be sufficient for the reader to foresee the general algorithm

presented in the remaining of the section. Let $\Box((p \cup q) \rightarrow \langle \rangle (q \rightarrow or))$ be an LTL formula and let $\varphi_1, \varphi_2, \dots, \varphi_{10}$ be its subformulae, in breadth-first order:

$$\begin{aligned}
\varphi_1 &= \Box((p \cup q) \rightarrow \langle \rangle (q \rightarrow or)), \\
\varphi_2 &= (p \cup q) \rightarrow \langle \rangle (q \rightarrow or), \\
\varphi_3 &= p \cup q, \\
\varphi_4 &= \langle \rangle (q \rightarrow or), \\
\varphi_5 &= p, \\
\varphi_6 &= q, \\
\varphi_7 &= q \rightarrow or, \\
\varphi_8 &= q, \\
\varphi_9 &= or, \\
\varphi_{10} &= r.
\end{aligned}$$

Given any finite trace $t = e_1 e_2 \dots e_n$ of n events, one can recursively define a matrix $s[1..n, 1..10]$ of Boolean values $\{0, 1\}$, with the meaning that $s[i, j] = 1$ iff $t_i \models \varphi_j$ as follows:

$$\begin{aligned}
s[i, 10] &= (r \in e_i) \\
s[i, 9] &= s[i + 1, 10] \\
s[i, 8] &= (q \in e_i) \\
s[i, 7] &= s[i, 8] \text{ implies } s[i, 9] \\
s[i, 6] &= (q \in e_i) \\
s[i, 5] &= (p \in e_i) \\
s[i, 4] &= s[i, 7] \text{ or } s[i + 1, 4] \\
s[i, 3] &= s[i, 6] \text{ or } (s[i, 5] \text{ and } s[i + 1, 3]) \\
s[i, 2] &= s[i, 3] \text{ implies } s[i, 4] \\
s[i, 1] &= s[i, 2] \text{ and } s[i + 1, 1],
\end{aligned}$$

for all $i < n$, where *and*, *or*, *implies* are ordinary Boolean operations and $==$ is the equality predicate, where $s[n, 1..10]$ are defined as below:

$$\begin{aligned}
s[n, 10] &= (r \in e_n) \\
s[n, 9] &= s[n, 10] \\
s[n, 8] &= (q \in e_n) \\
s[n, 7] &= s[n, 8] \text{ implies } s[n, 9] \\
s[n, 6] &= (q \in e_n) \\
s[n, 5] &= (p \in e_n) \\
s[n, 4] &= s[n, 7] \\
s[n, 3] &= s[n, 6] \\
s[n, 2] &= s[n, 3] \text{ implies } s[n, 4] \\
s[n, 1] &= s[n, 2].
\end{aligned}$$

Note again that the trace needs to be *traversed backwards*, and that the row n of s is filled according to the stationary view of finite traces in their last event. An important observation is that, like in many other dynamic programming algorithms, one does not have to store all the table $s[1..n, 1..10]$, which would be quite large in practice; in this case, one needs only two rows, $s[i, 1..10]$ and $s[i + 1, 1..10]$, which we shall write *now* and *next* from now on, respectively. It is now only a simple exercise to write up the following algorithm:

```

INPUT: trace  $t = e_1 e_2 \dots e_n$ 
 $next[10] \leftarrow (r \in e_n);$ 
 $next[9] \leftarrow next[10];$ 
 $next[8] \leftarrow (q \in e_n);$ 
 $next[7] \leftarrow next[8] \text{ implies } next[9];$ 
 $next[6] \leftarrow (q \in e_n);$ 
 $next[5] \leftarrow (p \in e_n);$ 
 $next[4] \leftarrow next[7];$ 
 $next[3] \leftarrow next[6];$ 
 $next[2] \leftarrow next[3] \text{ implies } next[4];$ 
 $next[1] \leftarrow next[2];$ 
for  $i = n - 1$  downto 1 do {
     $now[10] \leftarrow (r \in e_i);$ 
     $now[9] \leftarrow next[10];$ 
     $now[8] \leftarrow (q \in e_i);$ 
     $now[7] \leftarrow now[8] \text{ implies } now[9];$ 
     $now[6] \leftarrow (q \in e_i);$ 
     $now[5] \leftarrow (p \in e_i);$ 

```

```

    now[4] ← now[7] or next[4];
    now[3] ← now[6] or (now[5] and next[3]);
    now[2] ← now[3] implies now[4];
    now[1] ← now[2] and next[1];
    next ← now }
output(next[1]);

```

The algorithm above can be further optimized, noticing that only the bits 10, 4, 3 and 1 are needed in the vectors *now* and *next*, as we did for past time LTL in [88, 87]. The analysis of this algorithm is straightforward. Its time complexity is $\Theta(n \cdot m)$ while the memory required is $2 \cdot m$ bits, where n is the length of the trace and m is the size of the LTL formula.

8.2.2 Generating Dynamic Programming Algorithms

We now formally describe our algorithm that synthesizes dynamic programming algorithms like the one above from LTL formulae. Our synthesizer is generic, the potential user being expected to adapt it to his/her desired target language. The algorithm consists of three main steps:

Breadth First Search. The LTL formula should be first visited in breadth-first search (BFS) order to assign increasing numbers to subformulae as they are visited. Let $\varphi_1, \varphi_2, \dots, \varphi_m$ be the list of all subformulae in BFS order. Because of the semantics of finite trace LTL, this step ensures us that the truth value of $t_i \models \varphi_j$ can be completely determined from the truth values of $t_i \models \varphi_{j'}$ for all $j < j' \leq m$ and the truth values of $t_{i+1} \models \varphi_{j'}$ for all $j \leq j' \leq m$. This recurrence gives the order in which one should generate the code.

Loop Initialization. Before we generate the “for” loop, we should first initialize the vector *next*[1..*m*], which basically gives the truth values of the subformulae on the empty trace. According to the semantics of LTL, one should fill the vector *next* backwards. For a given $m \geq j \geq 1$, *next*[*j*] is calculated as follows:

- If φ_j is a variable then $\text{next}[j] = (\varphi_j \in e_n)$. In a more complex setting, where φ_j was a state predicate, one would have to evaluate φ_j in the final state in the execution trace;
- If φ_j is $!\varphi_{j'}$ for some $j < j' \leq m$, then $\text{next}[j] = \text{not } \text{next}[j']$, where *not* is the negation operation on Booleans (bits);

- If φ_j is $\varphi_{j_1} \text{ Op } \varphi_{j_2}$ for some $j < j_1, j_2 \leq m$, then $\text{next}[j] = \text{next}[j_1] \text{ op } \text{next}[j_2]$, where Op is any propositional operation and op is its corresponding Boolean operation;
- If φ_j is $\circ\varphi_{j'}$, $\Box\varphi_{j'}$, or $\langle\rangle\varphi_{j'}$, then clearly $\text{next}[j] = \text{next}[j']$ according to the stationary semantics of our finite trace LTL;
- If φ_j is $\varphi_{j_1} \cup \varphi_{j_2}$ for some $j < j_1, j_2 \leq m$, then $\text{next}[j] = \text{next}[j_2]$ for the same reason as above.

Loop Generation. Because of the dependences in the recursive definition of finite trace LTL satisfaction relation, one is expected to visit the remaining of the trace backwards, so the loop index will vary from $n - 1$ down to 1. The loop body will update/calculate the vector *now* and in the end will move it into the vector *next* to serve as basis for the next iteration. At a certain iteration i , the vector *now* is updated also backwards as follows:

- If φ_j is a variable then $\text{now}[j] = (\varphi_j \in e_i)$.
- If φ_j is $!\varphi_{j'}$ for some $j < j' \leq m$, then $\text{now}[j] = \text{not } \text{now}[j']$;
- If φ_j is $\varphi_{j_1} \text{ Op } \varphi_{j_2}$ for $j < j_1, j_2 \leq m$, then $\text{now}[j] = \text{now}[j_1] \text{ op } \text{now}[j_2]$, where Op is any propositional operation and op is its corresponding Boolean operation;
- If φ_j is $\circ\varphi_{j'}$ then $\text{now}[j] = \text{next}[j']$ since φ_j holds now if and only if $\varphi_{j'}$ held at the previous step (which processed the next event, the $i + 1$ -th);
- If φ_j is $\Box\varphi_{j'}$ then $\text{now}[j] = \text{now}[j']$ and $\text{next}[j]$ because φ_j holds now if and only if $\varphi_{j'}$ holds now and φ_j held at the previous iteration;
- If φ_j is $\langle\rangle\varphi_{j'}$ then $\text{now}[j] = \text{now}[j']$ or $\text{next}[j]$ because of similar reasons as above;
- If φ_j is $\varphi_{j_1} \cup \varphi_{j_2}$ for some $j < j_1, j_2 \leq m$, then $\text{now}[j] = \text{now}[j_2]$ or $(\text{now}[j_1] \text{ and } \text{next}[j])$.

After each iteration, $\text{next}[1]$ says whether the initial LTL formula is validated by the trace $e_i e_{i+1} \dots e_n$. Therefore, the desired output is $\text{next}[1]$ after the last iteration. Putting all the above together, one can now write up the generic pseudocode presented below which can be implemented very efficiently on any current platform. Since the BFS procedure is linear,

the algorithm synthesizes a dynamic programming algorithm from an LTL formula in linear time and of linear size with the size of the formula.

The following generic program implements the discussed technique. It takes as input an LTL formula and generates a “for” loop which traverses the trace of events backwards, thus validating or invalidating the formula.

```

INPUT: LTL formula  $\varphi$ 
output("INPUT: trace  $t = e_1e_2...e_n$ ");
let  $\varphi_1, \varphi_2, \dots, \varphi_m$  be all the subformulae of  $\varphi$  in BFS order
for  $j = m$  downto 1 do {
    output("next",  $j$ , "["  $\leftarrow$  ");
    if  $\varphi_j$  is a variable then output("( $\varphi_j, \in e_n$ );");
    if  $\varphi_j = !\varphi_{j'}$  then output("not next",  $j'$ , "[";");
    if  $\varphi_j = \varphi_{j_1} Op \varphi_{j_2}$  then output("next",  $j_1$ , "[" op next",  $j_2$ , "[";");
    if  $\varphi_j = \circ\varphi_{j'}$  then output("next",  $j'$ , "[";");
    if  $\varphi_j = \Box\varphi_{j'}$  then output("next",  $j'$ , "[";");
    if  $\varphi_j = \langle\rangle\varphi_{j'}$  then output("next",  $j'$ , "[";");
    if  $\varphi_j = \varphi_{j_1} \cup \varphi_{j_2}$  then output("next",  $j_2$ , "[";"); }
output("for  $i = n - 1$  downto 1 do {");
for  $j = m$  downto 1 do {
    output("now",  $j$ , "["  $\leftarrow$  ");
    if  $\varphi_j$  is a variable then output("( $\varphi_j, \in e_i$ );");
    if  $\varphi_j = !\varphi_{j'}$  then output("not now",  $j'$ , "[";");
    if  $\varphi_j = \varphi_{j_1} Op \varphi_{j_2}$  then output("now",  $j_1$ , "[" op now",  $j_2$ , "[";");
    if  $\varphi_j = \circ\varphi_{j'}$  then output("next",  $j'$ , "[";");
    if  $\varphi_j = \Box\varphi_{j'}$  then output("now",  $j'$ , "[" and next",  $j$ , "[";");
    if  $\varphi_j = \langle\rangle\varphi_{j'}$  then output("now",  $j'$ , "[" or next",  $j$ , "[";");
    if  $\varphi_j = \varphi_{j_1} \cup \varphi_{j_2}$  then output("now",  $j_2$ , "[" or (now",  $j_1$ , "["
and next",  $j$ , "[";"); }
output("next  $\leftarrow$  now; }");
output("output next[1];");

```

where *Op* is any propositional connective and *op* is its corresponding Boolean operator.

The Boolean operations used above are usually very efficiently implemented on any microprocessor and the vectors of bits *next* and *now* are small enough to be kept in cache. Moreover, the dependencies between

instructions in the generated “for” loop are simple to analyze, so a reasonable compiler can easily unfold or/and parallelize it to take advantage of machine’s resources. Consequently, the generated code is expected to run very fast.

The dynamic programming technique presented in this section is as efficient as one can hope, but, unfortunately, has a major drawback: it needs to traverse the execution trace backwards. From a practical perspective, that means that the instrumented program is run for some period of time while its execution trace is saved, and then, after the program was stopped, its execution trace is traversed backwards and (efficiently) analyzed. Besides the obvious inconvenience due to storing potentially huge execution traces, this method cannot be used to monitor programs synchronously.

8.3 A Forwards and Often Synchronous Algorithm

In this section we shall present a more efficient rewriting semantics for LTL, based on the idea of consuming the events in the trace, one by one, and updating a data structure (which is also a formula) corresponding to the effect of the event on the value of the formula. An important advantage of this algorithm is that it often detects when a formula is violated or validated before the end of the execution trace, so, unlike the algorithms above, it is suitable for online monitoring. Our decision to write an operational semantics this way was motivated by an attempt to program such an algorithm in Java, where such a solution would be natural. The presented rewriting-based algorithm is linear in the size of the execution trace and worst-case exponential in the size of the monitored LTL formula.

8.3.1 An Event Consuming Algorithm

We will implement this rewriting based algorithm by extending the definition of the event consuming operation $_ \{ _ \} : \text{Formula Event}^* \rightarrow \text{Formula}$ to temporal operators, with the following intuition. Assuming a trace E, T consisting of event E followed by trace T , a formula X holds on this trace if and only if $X\{E\}$ holds on the remaining trace T . If the event E is terminal then $X\{E \ * \}$ holds if and only if X holds under standard LTL semantics on the infinite trace containing only the event E .

```
fmod LTL-REVISED is
  protecting LTL .
```

```

vars X Y : Formula .
var E : Event .
var T : Trace .
eq (o X){E} = X .
eq (o X){E *} = X{E *} .
eq (<> X){E} = X{E} \ / <> X .
eq (<> X){E *} = X{E *} .
eq ([] X){E} = X{E} /\ [] X .
eq ([] X){E *} = X{E *} .
eq (X U Y){E} = Y{E} \ / X{E} /\ X U Y .
eq (X U Y){E *} = Y{E *} .

op _|-_ : Trace Formula -> Bool .
eq E |- X = [X{E *}] .
eq E,T |- X = T |- X{E} .
endfm

```

The rule for the temporal operator $[]X$ should be read as follows: the formula X must hold now ($X\{E\}$) and also in the future ($[]X$). The sub-expression $X\{E\}$ represents the formula that must hold on the rest of the trace in order for X to hold now.

As an example, consider again the traffic light controller safety formula $[](\text{green} \rightarrow \text{!red} \cup \text{yellow})$, which is first rewritten to $[](\text{true} ++ \text{green} ++ \text{green} /\ (\text{true} ++ \text{red}) \cup \text{yellow})$ by the equations in module PROP-CALC. This formula modified by an event `green yellow` (notice that two lights can be lit at the same time) yields the rewriting sequence

```

([](true ++ green ++ green /\ (true ++ red) U yellow)){green yellow} ==>
(true ++ green{green yellow}
  ++ green{green yellow} /\ ((true ++ red) U yellow){green yellow}
  /\ [](true ++ green ++ green /\ (true ++ red) U yellow) ==>
((true ++ red) U yellow){green yellow}
  /\ [](true ++ green ++ green /\ (true ++ red) U yellow) ==>
(yellow{green yellow} \ / ((true ++ red{green yellow}) /\ (true ++ red) U yellow)
  /\ [](true ++ green ++ green /\ (true ++ red) U yellow) ==>
[](true ++ green ++ green /\ (true ++ red) U yellow)

```

which is exactly the original formula, while the same formula transformed by just the event `green` yields

```

([](true ++ green ++ green /\ (true ++ red) U yellow)){green} ==>
(true ++ green{green}
  ++ green{green} /\ ((true ++ red) U yellow){green}
  /\ [](true ++ green ++ green /\ (true ++ red) U yellow) ==>

```



```

((true ++ red) U yellow){green}
  /\ [](true ++ green ++ green /\ (true ++ red) U yellow)      ==>
(yellow{green} \/ ((true ++ red{green}) /\ (true ++ red) U yellow)
  /\ [](true ++ green ++ green /\ (true ++ red) U yellow)      ==>
(true ++ red) U yellow /\ [](true ++ green ++ green /\ (true ++ red) U yellow)

```

which further modified by an event red yields

```

(yellow{red} \/ (true ++ red{red})) /\ (true ++ red) U yellow
  /\ ([](true ++ green ++ green /\ (true ++ red) U yellow)){red} ==>
false /\ ([](true ++ green ++ green /\ (true ++ red) U yellow)){red} ==>
false

```

When the current formula becomes `false`, as it happened above, we say that the original formula has been violated. Indeed, the current formula will remain `false` for any subsequent trace of events, so the result of the monitoring session will be `false`.

Note that the rewriting system described so far obviously terminates, because what it does is to propagate the current event to the atomic subformulae, replace those by either true or false, and eventually canonize the newly obtained formula.

Some operators could be defined in terms of others, as is typically the case in the standard semantics for LTL. For example, we could introduce an equation of the form: $\langle\!\rangle X = \text{true} \cup X$, and then eliminate the rewriting rule for $\langle\!\rangle X$ in the above module. This turns out to be less efficient in practice though, because more rewrites are needed. This happens regardless of whether one enables memoization (explained in detail in Subsection 8.3.3) or not, because memoization brings a real benefit only when previously processed terms are to be reduced again.

This module eventually defines a new satisfaction relation $_|- _$ between traces and formulae. The term $T \vdash X$ is evaluated now by an iterative traversal over the trace, where each event transforms the formula. Note that the new formula that is generated at each step is always kept small by being reduced to normal form via the equations in the `PROP-CALC` module in Subsection 2.1.2.

Our current JPAX implementation of the rewriting algorithm above executes the last two rules of the module `LTL-REVISED` outside the main rewriting engine. More precisely, Maude is started in its *loop mode* [49], which provides the capability of enabling rewriting rules in a reactive system style: a “state” term is stored, which can then be modified via rewriting rules that are activated by ASCII text events that are provided via the standard

I/O. JPAX starts a Maude process and assigns the formula to be monitored as its loop mode state. Then, as events are received from the monitored program, they are filtered and forwarded to the Maude module, which then enables rewriting on the term $X\{E\}$, where X is the current formula and E is the newly received event; the normal form of this reduction, a formula, is stored as the new loop mode state term. The process continues until the last event is received. JPAX tags the last event, asking Maude to reduce a term $X\{E \ast\}$; the result will be either `true` or `false`, which is reported to the user at the end of the monitoring session².

A natural question here is how big the stored formula can grow during a monitoring session. Such a formula will consist of Boolean combinations of sub-formulae of the initial formula, kept in a minimal canonical form. This can grow exponentially in the size of the initial formula in the worst-case (see [147] for a related result for extended regular expressions).

Theorem 15 *For any formula X of size m and any sequence of events to be monitored E_1, E_2, \dots, E_n , the formula $X\{E_1\}\{E_2\} \dots \{E_n\}$ needs $O(2^m)$ space to be stored. Moreover, the exponential space cannot be avoided: any synchronous or asynchronous forwards monitoring algorithm for LTL requires space $\Omega(2^{c\sqrt{m}})$ space, where c is some fixed constant.*

Proof: Due to the Boolean ring simplification rules in PROP-CALC, any LTL formula is kept in a canonical form, which is an exclusive disjunction of conjunctions, where conjuncts have temporal operators at top. Moreover, after processing any number of events, in our case E_1, E_2, \dots, E_n , the conjuncts in the normal form of $X\{E_1\}\{E_2\} \dots \{E_n\}$ are subterms of the initial formula X , each having a temporal operator at its top. Since there are at most m such subformulae of X , it follows that there are at most 2^m possibilities to combine them in a conjunction. Therefore, one needs space $O(2^m)$ to store any exclusive disjunction of such conjunctions. This reasoning only applies on “idealistic” rewriting engines, which carefully optimize space needs during rewriting. It is not clear to us whether Maude is able to attain this space upper bound in all situations.

For the space lower bound of any finite trace LTL monitoring algorithm, consider a simplified framework with only two atomic predicate and therefore only four possible states. For simplicity, we encode these four states by 0, 1,

²In fact, JPAX reports a similar message also when the current monitoring requirement becomes `true` or `false` at any time during the monitoring process.

$\#$ and $\$$. Consider also some natural number k and the language

$$L_k = \{\sigma \# w \# \sigma' \$ w \mid w \in \{0, 1\}^k \text{ and } \sigma, \sigma' \in \{0, 1, \#\}^*\}.$$

This language was previously used in several works [119, 120, 147] to prove lower bounds. The language can be shown to contain exactly those finite traces satisfying the following LTL formula [120] of size $\Theta(k^2)$:

$$\phi_k = [(!\$)\mathcal{U}(\$ \wedge \circ \square(!\$))] \wedge \diamond[\# \wedge \circ^{n+1} \# \wedge \bigwedge_{i=1}^n ((\circ^i 0 \wedge \square(\$ \rightarrow \circ^i 0)) \vee (\circ^i 1 \wedge \square(\$ \rightarrow \circ^i 1)))].$$

Let us define an equivalence relation on finite traces in $(0 + 1 + \#)^*$. For a $\sigma \in (0 + 1 + \#)^*$, define $S(\sigma) = \{w \in (0 + 1)^k \mid \exists \lambda_1, \lambda_2. \lambda_1 \# w \# \lambda_2 = \sigma\}$. We say that $\sigma_1 \equiv_k \sigma_2$ if and only if $S(\sigma_1) = S(\sigma_2)$. Now observe that the number of equivalence classes of \equiv_k is 2^{2^k} ; this is because for any $S \subseteq (0 + 1)^k$, there is a σ such that $S(\sigma) = S$.

Since $|\phi_k| = \Theta(k^2)$, it follows that there is some constant c' such that $|\phi_k| \leq c'k^2$ for all large enough k . Let c be the constant $1/\sqrt{c'}$. We will prove this lower bound result by contradiction. Suppose \mathcal{A} is an LTL forwards monitoring algorithm that uses less than $2^{c\sqrt{m}}$ space for any LTL formulae of large enough size m . We will look at the behavior of the algorithm \mathcal{A} on inputs of the form ϕ_k . So $m = |\phi_k| \leq c'k^2$, and \mathcal{A} uses less than 2^k space. Since the number of equivalence classes of \equiv_k is 2^{2^k} , by the pigeon hole principle there must be two strings $\sigma_1 \not\equiv_k \sigma_2$ such that the memory of \mathcal{A} on ϕ_k after reading $\sigma_1 \$$ is the same as the memory after reading $\sigma_2 \$$. In other words, \mathcal{A} running on ϕ_k will give the same answer on all traces of the form $\sigma_1 \$w$ and $\sigma_2 \$w$. Now since $\sigma_1 \not\equiv_k \sigma_2$, it follows that $(S(\sigma_1) \setminus S(\sigma_2)) \cup (S(\sigma_2) \setminus S(\sigma_1)) \neq \emptyset$. Take $w \in (S(\sigma_1) \setminus S(\sigma_2)) \cup (S(\sigma_2) \setminus S(\sigma_1))$. Then clearly, exactly one out of $\sigma_1 \$w$ and $\sigma_2 \$w$ is in L_k , and so \mathcal{A} running on ϕ_k gives the wrong answer on one of these inputs. Therefore, \mathcal{A} is not a correct. \square

It seems, however, that this worst-case exponential complexity in the size of the LTL formula is more of theoretical importance than practical, since in general the size of the formula rarely grew more than twice in our experiments. Verification results are very encouraging and show that this optimized semantics is orders of magnitude faster than the first semantics. Traces of less than 10,000 events are verified in milliseconds, while traces of 100,000 events never needed more than 3 seconds. This technique scales quite well; we were able to monitor even traces of hundreds of millions events. As a concrete example, we created an artificial trace by repeating 10 million times the 10 event trace in Subsection 8.1.2, and then checked

it against the formula $\square(\text{green} \rightarrow !\text{red} \cup \text{yellow})$. There were needed 4.9 billion rewriting steps for a total of about 1,500 seconds. In Subsection 8.3.3 we will see how this algorithm can be made even more efficient, using memoization.

8.3.2 Correctness and Completeness

In this subsection we prove that the algorithm presented above is correct and complete with respect to the semantics of finite trace LTL presented in Section 8.1. The proof is done completely in Maude, but since Maude is not intended to be a theorem prover, we actually have to generate the proof obligations by hand. In other words, the proof that follows was *not* generated automatically. However, it could have been mechanized by using proof assistants and/or theorem provers like KUMO [75], PVS [166], or Maude-ITP [42]. We have already done it in PVS, but we prefer to use only plain Maude in this paper.

Theorem 16 *For any trace T and any formula X , $T \models X$ if and only if $T \vdash X$.*

Proof: By induction, both on traces and formulae. We first need to prove two lemmas, namely that the following two equations hold in the context of both LTL and LTL-REVISED:

$$\begin{aligned} (\forall E : \text{Event}, X : \text{Formula}) \quad E \models X &= E \vdash X, \\ (\forall E : \text{Event}, T : \text{Trace}, X : \text{Formula}) \quad E, T \models X &= T \models X\{E\}. \end{aligned}$$

We prove them by structural induction on the formula X . Constants e and x are needed in order to prove the first lemma via the theorem of constants. However, since we prove these lemmas by structural induction on X , we not only have to add two constants e and t for the universally quantified variables E and T , but also two other constants y and z standing for formulas which can be combined via operators to give other formulas. The induction hypotheses are added to the following specification via equations. Notice that we merged the two proofs to save space. A proof assistant like KUMO, PVS or Maude-ITP would prove them independently, generating only the needed constants for each of them.

```
fmod PROOF-OF-LEMMAS is
  extending LTL .
  extending LTL-REVISED .
```

```

op e : -> Event .
op t : -> Trace .
ops a b c : -> Atom .
ops y z : -> Formula .
eq e |= y = e |- y .
eq e |= z = e |- z .
eq e,t |= y = t |= y{e} .
eq e,t |= z = t |= z{e} .
eq b{e} = true .
eq c{e} = false .
endfm

```

It is worth reminding the reader at this stage that the functional modules in Maude have initial semantics, so proofs by induction are valid. Before proceeding further, the reader should be aware of the operational semantics of the operation `_==_`, namely that the two argument terms are first reduced to their normal forms which are then compared syntactically (but modulo associativity and commutativity); it returns `true` if and only if the two normal forms are equal. Therefore, the answer `true` means that the two terms are indeed semantically equal, while `false` only means that they could not be proved equal; they can still be equal.

```

reduce (e |= a      == e |- a)
  and (e |= true    == e |- true)
  and (e |= false   == e |- false)
  and (e |= y /\ z == e |- y /\ z)
  and (e |= y ++ z == e |- y ++ z)
  and (e |= [] y    == e |- [] y)
  and (e |= <> y     == e |- <> y)
  and (e |= y U z   == e |- y U z)
  and (e |= o y     == e |- o y)

  and (e,t |= true  == t |= true{e})
  and (e,t |= false == t |= false{e})
  and (e,t |= b     == t |= b{e})
  and (e,t |= c     == t |= c{e})
  and (e,t |= y /\ z == t |= (y /\ z){e})
  and (e,t |= y ++ z == t |= (y ++ z){e})
  and (e,t |= [] y   == t |= ([] y){e})
  and (e,t |= <> y   == t |= (<> y){e})
  and (e,t |= y U z  == t |= (y U z){e})
  and (e,t |= o y    == t |= (o y){e}) .

```

It took Maude 129 reductions to prove these lemmas. Therefore, one can

safely add now these lemmas as follows:

```
fmod LEMMAS is
  protecting LTL .
  protecting LTL-REVISED .
  var E : Event .
  var T : Trace .
  var X : Formula .
  eq E |= X = E |- X .
  eq E,T |= X = T |- X{E} .
endfm
```

We can now prove the theorem, by induction on traces. More precisely, we show:

$\mathcal{P}(E)$, and
 $\mathcal{P}(T)$ implies $\mathcal{P}(E,T)$, for all events E and traces T ,

where $\mathcal{P}(T)$ is the predicate “for all formulas X , $T \models X$ iff $T \vdash X$ ”. This induction schema can be easily formalized in Maude as follows:

```
fmod PROOF-OF-THEOREM is
  protecting LEMMAS .
  op e : -> Event .
  op t : -> Trace .
  op x : -> Formula .
  var X : Formula .
  eq t |= X = t |- X .
endfm

reduce e |= x == e |- x .
reduce e,t |= x == e,t |- x .
```

Notice the difference in role between the constant x and the variable X . The first reduction proves the base case of the induction, using the theorem of constants for the universally quantified variable X . In order to prove the induction step, we first applied the theorem of constants for the universally quantified variables E and T , then added $\mathcal{P}(t)$ to the hypothesis (the equation “ $\text{eq } t \models X = t \vdash X$.”), and then reduced $\mathcal{P}(e \ t)$ using again the theorem of constants for the universally quantified variable X . Like in the proofs of the lemmas, we merged the two proofs to save space. \square

8.3.3 Further Optimization by Memoization

Even though the formula transforming algorithm in Subsection 8.3.1 can process 100 million events in about 25 minutes, which is relatively reasonable for practical purposes, it can be significantly improved by adding only 5 more characters to the existing Maude code presented so far. More precisely, one can replace the operation declaration

```
op _{_} : Formula Event* -> Formula [prec 10]
```

in module LOGICS-BASIC by the operation declaration

```
op _{_} : Formula Event* -> Formula [memo prec 10]
```

The attribute `memo` added to an operation declaration instructs Maude to memorize, or cache, the normal forms of terms rooted in that operation, i.e., those terms will be rewritten only once. Memoization is implemented by hashing, where the entry in the hash table is given by the term to be reduced and the value in the hash is its normal form. In our concrete example, memoization has the effect that any LTL formula will be transformed by a given event exactly once during the monitoring sequence; if the same formula and the same event occur in the future, the resulting modified formula is extracted from the hash table without applying any rewriting step. If one thinks of LTL in terms of automata, then our new algorithm corresponds to building the monitoring automaton *on the fly*. The obvious benefit of this technique is that only the *needed* part of the automaton is built, namely that part that is reachable during monitoring a particular sequence of events, which is practically very useful because the entire automaton associated to an LTL formula can be exponential in size, so storing it might become a problem.

The use of memoization brings a significant improvement in the case of LTL. For example, the same sequence of 100 million events, which took 1500 seconds using the algorithm presented in Subsection 8.3.1, takes only 185 seconds when one uses memoization, for a total of 2.2 rewritings per processed event and 540,000 events processed per second! We find these numbers amazingly good for any practical purpose we can think of and believe that, taking into account the simplicity, obvious correctness and elegance of the rewriting based algorithm (implemented basically by 8 rewriting rules in LTL-REVISED), it would be hard to argue for any other implementation of LTL monitoring. One should, however, be careful when one uses memoization because hashing slows down the rewriting engine. LTL

is a happy case where memoization brings a significant improvement, because the operational semantics of all the operators can be defined recursively, so formulae repeat often during the monitoring process. However, there might be monitoring logics where memoization could be less efficient. Such a logic would probably be an extension of LTL with time, allowing formulae of the form “ $\langle 5 \rangle X$ ” with the meaning “eventually in 5 units of time X ”, because of a potentially very large number of terms to be memoized: $\langle 5 \rangle X$, $\langle 4 \rangle X$, etc. Experimentation is certainly needed if one designs a new logic for monitoring and wants to use memoization.

8.4 Generating Forwards, Synchronous and Efficient Monitors

Even though the rewriting based monitoring algorithm presented in the previous section performs quite well in practice, there can be situations in which one wants to minimize the monitoring overhead as much as possible. Additionally, despite its simplicity and elegance, the procedure above requires an efficient rewriting engine modulo associativity and commutativity, which may not be available or may not be desirable on some monitoring platforms, such as, for example, within an embedded system.

In this section we give a technique, based on ideas presented previously in the paper, to generate automata-based optimal monitors for future time LTL formulae. By optimality we here mean everything one may expect, such as minimal number of states, forwards traversal of execution traces, synchronicity, efficiency, but also less standard optimality features, such as transiting from one state to another with a minimum amount of computation. In order to effectively do this we introduce the notion of *binary transition tree* (BTT), as a generalization of binary decision diagrams (BDD) [29], whose purpose is to provide an *optimal order* in which state predicates need to be evaluated to decide the next state. The motivation for this is that in practical applications evaluating a state predicate is a time consuming task, such as for example to check whether a vector is sorted. The associated finite state machines are called *binary transition tree finite state machines* (BTT-FSM).

The drawback of generating an optimal BTT-FSM statically, i.e., before monitoring, is the worst-case double exponential time/space required at startup. Therefore, the algorithm presented in this section is recommended for situations where the LTL formulae to monitor are relatively small in size

but the runtime overhead is desired to be minimal. It is worth noting that the BTT-FSM generation process can potentially take place on a machine different from the one performing the monitoring. In particular, one can think of a WWW fast server offering LTL-to-BTT-FSM services via the Internet, which can also maintain a database of already generated BTT-FSMs to avoid regenerating the same monitors.

8.4.1 From LTL Formulae to BTT-FSMs

Informally, our algorithm to generate optimal BTT-FSMs from LTL formulae consists of two steps. First, it generates an MT-FSM with a minimum number of states. Then, using the technique presented in the previous subsection, it generates an optimal BTT from each MT.

To generate a minimal MT-FSM, our algorithm uses the rewriting based procedure presented in Section 8.3 on all possible events, until the set of formulae to which the original LTL formula can “evolve” stabilizes. The procedure LTL2MT-FSM shown in Figure 8.1 builds an MT-FSM whose states are formulae, with the help of a validity checker. Initially, the set S of states contains only the original LTL formula. LTL2MT-FSM is called on the original LTL formula, and then recursively in a depth-first manner on all the formulae to which the initial formula can ever evolve via the event-consuming operator $_{-}\{_ \}$ introduced in Section 8.3.

For each LTL state formula φ in S , multi-transitions $\mu(\varphi)$ and $\mu^*(\varphi)$ are maintained. For each possible event θ , one first updates $\mu^*(\varphi)$ by considering the case in which θ is the last event in a trace (step 8), and then the current formula φ evolves into the corresponding formula φ_θ (step 9). If some equivalent formula φ' to φ_θ has already been discovered then one only needs to modify the multi-transition set of φ accordingly in order to point to φ' (step 11). Notice that equivalence of LTL formulae is checked by using a validity procedure (step 10), which is given in Figure 8.2. If there is no formula in S equivalent to φ_θ , then the new formula φ_θ is added to S , multi-transition $\mu(\varphi)$ is updated accordingly, and then the MT-FSM generation procedure is called recursively on φ_θ . This way, one eventually generates all possible LTL formulae into which the initial formula can ever evolve during a monitoring session; this happens, of course, modulo finite trace LTL semantic equivalence, implemented elegantly using the validity checker described below. By Theorem 15, this recursion will eventually terminate, leading to an MT-FSM*.

The procedure VALID used in LTL2MT-FSM above is defined in Figure

```

1. let  $S$  be  $\varphi$ 
2. procedure LTL2MT-FSM( $\varphi$ )
3.   let  $\mu^*(\varphi)$  be  $\emptyset$ 
4.   let  $\mu(\varphi)$  be  $\emptyset$ 
5.   foreach  $\theta : A \rightarrow \{true, false\}$  do
6.     let  $e_\theta$  be the list of atoms  $a$  with  $\theta(a) = true$ 
7.     let  $p_\theta$  be the proposition  $\bigwedge\{a \mid \theta(a) = true\} \wedge \bigwedge\{\neg a \mid \theta(a) = false\}$ 
8.     let  $\mu^*(\varphi)$  be MERGE( $[p_\theta ? \varphi\{e_\theta^*\}], \mu^*(\varphi)$ )
9.     let  $\varphi_\theta$  be  $\varphi\{e_\theta\}$ 
10.    if there is  $\varphi' \in S$  with  $\text{VALID}(\varphi_\theta \leftrightarrow \varphi')$ 
11.      then let  $\mu(\varphi)$  be MERGE( $[p_\theta ? \varphi']$ ,  $\mu(\varphi)$ )
12.    else let  $S$  be  $S \cup \{\varphi_\theta\}$ 
13.      let  $\mu(\varphi)$  be MERGE( $[p_\theta ? \varphi_\theta]$ ,  $\mu(\varphi)$ )
14.      LTL2MT-FSM( $\varphi_\theta$ )
15.    endfor
16.    if  $\mu(\varphi) = [true ? \varphi]$  and  $\mu^*(\varphi) = [true ? b]$  then replace  $\varphi$  by  $b$  everywhere
17. endprocedure

```

Figure 8.1: Algorithm to generate a minimal MT-FSM* $(S, A, \mu, \mu^*, \varphi)$ from an LTL formula φ .

8.2. It essentially follows the same idea of generating all possible formulae to which the original LTL formula tested for validity can evolve via the event consumption operator defined by rewriting in Section 8.3, but for each newly obtained formula φ and for each event θ , it also checks whether an execution trace stopping at that event would be a rejecting sequence. The intuition for this check is that a formula is valid under finite trace LTL semantics if and only if any (finite) sequence of events satisfies that formula; since any generated formula corresponds to one into which the initial formula can evolve, we need to make sure that each of these formulae becomes *true* under any possible last monitored event. This is done by rewriting the term $\varphi\{e_\theta^*\}$ with the rules in Section 8.3 to its normal form (step 5.), i.e., *true* or *false*. The formula is valid if and only if there is no rejecting sequence; the entire space of evolving formulae is again explored by depth-first search. Notice that VALID does not test for equivalence of formulae, so it can potentially generate a larger number of formulae than LTL2MT-FSM. However, by

```

1. let  $S$  be  $\varphi$ 
2. function VALID( $\varphi$ )
3.   foreach  $\theta : A \rightarrow \{true, false\}$  do
4.     let  $e_\theta$  be the list of atoms  $a$  with  $\theta(a) = true$ 
5.     if  $\varphi\{e_\theta^*\} = false$  then return false
6.     let  $\varphi_\theta$  be  $\varphi\{e_\theta\}$ 
7.     if  $\varphi_\theta \notin S$ 
8.       then let  $S$  be  $S \cup \{\varphi_\theta\}$ 
9.       if VALID( $\varphi_\theta$ ) = false then return false
10.   endfor
11.   return true
12. end function

```

Figure 8.2: Validity checker for an LTL formula φ .

Theorem 15, this procedure will also eventually terminate.

Theorem 17 *Given an LTL formula φ of size m , the following hold:*

1. *The procedure VALID is correct, that is, VALID(φ) returns true if and only if φ is satisfied, as defined in Section 8.1, by any finite trace;*
2. *The space and time complexity of VALID(φ) is $2^{O(2^m)}$;*

Additionally, letting M denote the MT-FSM $(S, A, \mu, \mu^*, \varphi)$ generated by LTL2MT-FSM(φ), the following hold:*

3. *LTL2MT-FSM(φ) is correct; more precisely, M has the property that for any events $\theta_1, \dots, \theta_n, \theta$, it is the case that $\varphi \xrightarrow{\theta_1} \varphi_1 \dots \xrightarrow{\theta_n} \varphi_n \xrightarrow{\theta^*} true$ if and only if the finite trace $e_{\theta_1} \dots e_{\theta_n} e_\theta$ satisfies φ , and $\varphi \xrightarrow{\theta_1} \varphi_1 \dots \xrightarrow{\theta_n} \varphi_n \xrightarrow{\theta^*} false$ if and only if $e_{\theta_1} \dots e_{\theta_n} e_\theta$ does not satisfy φ ;*
4. *M is synchronous; more precisely, for any events $\theta_1, \dots, \theta_n$, it is the case that $\varphi \xrightarrow{\theta_1} \varphi_1 \dots \xrightarrow{\theta_n} \varphi_n \xrightarrow{e_\theta} true$ if and only if $e_{\theta_1} \dots e_{\theta_n} e_\theta$ is a valid prefix of φ , and $\varphi \xrightarrow{\theta_1} \varphi_1 \dots \xrightarrow{\theta_n} \varphi_n \xrightarrow{e_\theta} false$ if and only if $e_{\theta_1} \dots e_{\theta_n} e_\theta$ is a bad prefix of φ ;*
5. *The space and time complexity of LTL2MT-FSM(φ) is $2^{O(2^m)}$;*

6. M is the smallest MT-FSM* which is correct and synchronous (as defined in 3 and 4 above);
7. Monitoring against a BTT-FSM* corresponding to M , as shown in Subsection 6.1.2, needs $O(2^m)$ space and time.

Proof: 1. Using a depth-first search strategy, the procedure VALID visits all possible formulae into which the original formula φ can evolve via any possible sequence of events. These formulae are different modulo the rewriting rules in Section 8.3 which are used to simplify LTL formulae and to remove the derivatives, but those rules were shown to be sound, so VALID indeed explores all possible LTL formulae into which φ can *semantically* evolve. Moreover, for each such formula, say φ' , and each event, VALID checks at Step 5 whether a trace terminating with that event in φ' would lead to rejection. VALID(φ) returns true if and only if there is no such rejection, so it is indeed correct: if it had been some finite trace that did not satisfy φ then VALID would have found it during its exhaustive search.

2. The space required by VALID(φ) is clearly dominated by the size of S . By Theorem 15, each formula φ' into which φ can evolve needs $O(2^m)$ space to be stored. That means that there can be at most $2^{O(2^m)}$ such formulae. So the total space needed to store S in the worst case is $2^{O(2^m)}$. An amortized analysis of its running time, tells us that VALID runs its “for each event” loop one per formula in S . Since the number of events is much less than 2^m and since reducing the derivative of a formula by an event takes time proportional with the size of the formula, we deduce that the total running time of the VALID procedure is $2^{O(2^m)}$.

3. By Theorem 16 and the event consuming procedure described in Subsection 8.3.1, it follows that $e_{\theta_1} \dots e_{\theta_n} e_\theta$ satisfies φ if and only if $\varphi\{e_{\theta_1}\}\{\dots\}\{e_{\theta_n}\}\{e_\theta^*\}$ reduces to *true*, and that $e_{\theta_1} \dots e_{\theta_n} e_\theta$ does not satisfy φ if and only if $\varphi\{e_{\theta_1}\}\{\dots\}\{e_{\theta_n}\}\{e_\theta^*\}$ reduces to *false*. It is easy to see that VALID($\psi \leftrightarrow \psi'$) returns true if and only if ψ and ψ' are formulae equivalent under the finite trace LTL semantics. That means the MT-FSM* generated by LTL2MT-FSM(φ) contains a formula-state φ_1 which is equivalent to $\varphi\{e_{\theta_1}\}$; moreover, $\varphi \xrightarrow{\theta_1} \varphi_1$ in this MT-FSM*. Inductively, one can show that there is a series of formulae-states $\varphi_1, \dots, \varphi_n$ such that $\varphi \xrightarrow{\theta_1} \varphi_1 \dots \xrightarrow{\theta_n} \varphi_n$ is a sequence of transitions in the generated MT-FSM and $\varphi\{e_{\theta_1}\}\{\dots\}\{e_{\theta_n}\}$ is equivalent to φ_n . The rest follows by noting that $\varphi_n\{e_\theta^*\}$ reduces to true if and only if

$\varphi_n \xrightarrow{\theta^*} \text{true}$ in the generated MT-FSM, and that $\varphi_n\{e_\theta^*\}$ reduces to false if and only if $\varphi_n \xrightarrow{\theta^*} \text{false}$ in the MT-FSM.

4. As in 3 above, one can inductively show that there is a series of formulae $\varphi_1, \dots, \varphi_n, \varphi'$ such that $\varphi \xrightarrow{\theta_1} \varphi_1 \dots \xrightarrow{\theta_n} \varphi_n \xrightarrow{\theta} \varphi'$ is a sequence of transitions in the generated MT-FSM and $\varphi\{e_{\theta_1}\}\{\dots\}\{e_{\theta_n}\}\{e_\theta\}$ is equivalent to φ' . By Proposition 14, due to the use of the validity checker in step 10 of LTL2MT-FSM it follows that $e_{\theta_1} \dots e_{\theta_n} e_\theta$ is a valid prefix of φ if and only if φ' is equivalent to *true*, and that $e_{\theta_1} \dots e_{\theta_n} e_\theta$ is a bad prefix of φ if and only if φ' is equivalent to *false*. The rest follows now by Step 16 in the algorithm in Figure 8.1, which, due to the validity checker, provides a necessary and sufficient condition for a processed formula to be semantically equivalent to *true* or *false*, respectively.

5. The space required by LTL2MT-FSM(φ) is again dominated by the size of S . Like in the analysis of VALID in 2 above, by Theorem 15 we get that there can be at most $2^{O(2^m)}$ formulae generated in S , so the total space needed to store S in the worst case is also $2^{O(2^m)}$. For each newly added formula in S and for each event, Step 10 calls the procedure VALID potentially once for each already existing formula in S . It is important to notice that the formulae on which VALID is called are exclusive disjunctions of conjunctions of sub-formulae rooted in temporal operators of the original formula φ , so its space and time complexity will also be $2^{O(2^m)}$ each time it is called by LTL2MT-FSM. One can easily see now that the space and time requirements of LTL2MT-FSM(φ) are also $2^{O(2^m)}$ (the constant in the exponent $O(2^m)$ can be appropriately enlarged).

6. For any MT-FSM* machine $M' = (S', A, \mu', \mu'^*, q_0)$, one can associate a formula, more precisely a state in M , to each state in S' as follows. φ is associated to the initial state q_0 . Then for each transition $q_1 \xrightarrow{\theta} q_2$ in M' such that q_1 has a formula φ_1 associated and q_2 has no formula associated, then one associates that φ_2 to q_2 with the property that $\varphi_1 \xrightarrow{\theta} \varphi_2$ is a transition in M . For a state q in S' let φ_q be the formula in S associated to it. Since M' is correct and synchronous for φ , it follows that $\mathcal{L}_{M'}(q) = \mathcal{L}_M(\varphi_q)$ for any state q in S' . We can now show that the map associating a formula in S to any state in S' is surjective, which shows that M has therefore a minimal number of states. Let φ' be a formula in S and let $\varphi \xrightarrow{\theta_1} \varphi_1 \dots \xrightarrow{\theta_n} \varphi_n \xrightarrow{\theta} \varphi'$ be a sequence of transitions in M leading to φ' ; note that all formulae in S

are reachable via transitions in M from φ . Let q' be the state in S' such that $q_0 \xrightarrow{\theta_1} q_1 \cdots \xrightarrow{\theta_n} q_n \xrightarrow{\theta} q'$. Then $\mathcal{L}_{M'}(q') = \mathcal{L}_M(\varphi_{q'})$. Since by Proposition 14, $\mathcal{L}_{M'}(q') = \{t \mid e_{\theta_1} \dots e_{\theta_n} e_{\theta} t \in \mathcal{L}_{M'}(q_0)\}$ and $\mathcal{L}_M(\varphi') = \{t \mid e_{\theta_1} \dots e_{\theta_n} e_{\theta} t \in \mathcal{L}_M(\varphi)\}$, it follows that $\mathcal{L}_M(\varphi_{q'}) = \mathcal{L}_M(\varphi')$, that is, that $\varphi_{q'}$ and φ' are equivalent. Since Step 10 of the LTL2MT-FSM eventually uses the validity checker on any pairs of formulae in S , it follows that $\varphi_{q'} = \varphi'$.

7. In order to distinguish N pieces of data, $\log(N)$ bits are needed to encode each datum. Therefore, one needs $O(2^m)$ bits to encode a state of M , which is the main memory needed by the monitoring algorithm. Like in Theorem 15, we assume “idealistic” rewriting engines able to optimize the space requirements; we are not aware whether Maude is able to attain this performance or not. To make a transition from one state to another, a BTT-FSM* associated to the MT-FSM* generated by LTL2MT-FSM(φ) needs to only evaluate at most all the atomic predicates occurring in the formula, which, assuming that evaluating atom predicates takes unit time, is clearly $O(2^m)$. When the entire BTT is evaluated, it finally has to store the newly obtained state of the BTT-FSM*, which can reuse the space of the previous one, $O(2^m)$, and which also takes time $O(2^m)$. \square

Once the procedure LTL2MT-FSM terminates, the formulae φ , φ' , etc., are not needed anymore, so one can and should replace them by unique labels in order to reduce the amount of storage needed to encode the MT-FSM* and/or the corresponding BTT-FSM*. This algorithm can be relatively easily implemented in any programming language. We have, however, found Maude again a very elegant system for this task, implementing the entire LTL formula to BTT-FSM algorithm in about 200 lines of code.

8.4.2 Examples

The BTT-FSM generation algorithm presented in this section, despite its overall worst-case high startup time, can be very useful when formulae are relatively short, as it is most often the case in practice. For the traffic light controller requirement formula discussed previously in the paper, $\Box(\text{green} \rightarrow (!\text{red}) \cup \text{yellow})$, this algorithm generates in about 0.2 seconds the optimal BTT-FSM* in Figure 8.3, also shown in Figure 6.3 in flowchart notation; Figure 6.2 shows its optimal MT-FSM*. For simplicity, the states *true* and *false* do not appear in Figure 8.3. Notice that the atomic predicate **red** does *not* need to be evaluated on terminal events and that **green** does not need to be evaluated in state 2. In this example, the colors are not supposed to

State	Non-terminal event	Terminal event
1	yellow ? 1 : green ? red ? false : 2 : 1	yellow ? true : green ? false : true
2	yellow ? 1 : red ? false : 2	yellow ? true : false

Figure 8.3: An optimal BTT-FSM* for the formula $\square(\text{green} \rightarrow !\text{red} \cup \text{yellow})$.

exclude each other, that is, the traffic controller can potentially be both green and red.

The LTL formulae on which our algorithm has the worst performance are those containing many nested temporal operators (which are not frequently used in specifications anyway, because of the high risk of getting them wrong). For example, it takes our Maude implementation of this algorithm 1.3 seconds to generate the minimal 3-state (*true* and *false* states are not counted) BTT-FSM* for the formula $a \cup (b \cup (c \cup d))$ and 13.2 seconds to generate the 7-state minimal BTT-FSM* for the formula $((a \cup b) \cup c) \cup d$. It never took our current implementation more than a few seconds to generate the BTT-FSM* of any LTL formula of interest for our applications, i.e., non-artificial. Figure 8.4 shows the generated BTT-FSM of some artificial LTL formulae, taking together less than 15 seconds to be generated. To keep the figure small, the states *true* and *false* together with their self-transitions are not shown in Figure 8.4, and they are replaced by **t** and **f** in BTTs.

The generated BTT-FSM*s are monitored most efficiently on RAM machines, due to the fact that conditional statements are implemented via jumps in memory. Monitoring BTT-FSM*s using rewriting does not seem appropriate because it would require linear time, as a function of number of states, to extract the BTT associated to a state in a BTT-FSM*. However, we believe that the algorithm presented in Section 8.3 is satisfactory in practice if one is willing to use a rewriting engine for monitoring.

8.5 Conclusions

This paper presented a foundational study in using rewriting in runtime verification and monitoring of systems. After a short discussion on types of monitoring and mathematical and technological preliminaries, a finite trace linear temporal logic was defined, together with an immediate but inefficient implementation of a monitor following directly its semantics. Then an

Formula	State	Monitoring BTT	Terminating BTT
$\Box \diamond a$	1	1	$a ? t : f$
$\diamond(\Box a \vee \Box \neg a)$	—	—	—
$\Box(a \rightarrow \diamond b)$	1 2	$a ? (b ? 1 : 2) : 1$ $b ? 1 : 2$	$a ? (b ? t : f) : t$ $b ? t : f$
$a \mathcal{U} (b \mathcal{U} c)$	1 2	$c ? t : (a ? 1 : (b ? 2 : f))$ $c ? t : (b ? 2 : f)$	$c ? t : f$ $c ? t : f$
$a \mathcal{U} (b \mathcal{U} (c \mathcal{U} d))$	1 2 3	$d ? t : a ? 1 : b ? 2 : c ? 3 : f$ $d ? t : b ? 2 : c ? 3 : f$ $d ? t : c ? 3 : f$	$d ? t : f$ $d ? t : f$ $d ? t : f$
$((a \mathcal{U} b) \mathcal{U} c) \mathcal{U} d$	1 2 3 4 5 6 7	$d ? t : c ? 1 : b ? 4 : a ? 5 : f$ $b ? c ? t : 7 : a ? c ? 6 : 2 : f$ $b ? d ? t : c ? 1 : 4 : a ? d ? 6 : c ? 3 : 5 : f$ $c ? d ? t : 1 : b ? d ? 7 : 4 : a ? d ? 2 : 5 : f$ $b ? d ? c ? t : 7 : c ? 1 : 4 : a ? d ? c ? 6 : 2 : c ? 3 : 5 : f$ $b ? t : a ? 6 : f$ $c ? t : b ? 7 : a ? 2 : f$	$d ? t : f$ $c ? b ? t : f : f$ $d ? b ? t : f : f$ $d ? c ? t : f : f$ $d ? c ? b ? t : f : f : f$ $b ? t : f$ $c ? t : f$

Figure 8.4: Six BTT-FSM*s generated in less than 15 seconds.

efficient but ineffective implementation based on dynamic programming was presented, which traverses the execution trace backwards. The first effective and relatively efficient rewriting algorithm was further introduced, based on the idea of transforming the monitoring requirements as events are received from the monitored program. A non-trivial improvement of this algorithm based on hashing rewriting results, thereby reducing the number of rewritings performed during trace analysis, was also proposed. The hashing corresponds to building an observer automaton on-the-fly, having the advantage that only the part of the automaton that is needed for analyzing a given trace is generated. The resulting algorithm is very efficient. Since in many cases one would want to generate an observer finite state machine (or automaton) a priori, for example when a rewriting system cannot be used for monitoring, or when minimal runtime overhead is needed, a specialized data-structure called a binary transition tree (BTT) and corresponding finite state machines were introduced, and an algorithm for generating minimal such monitors from temporal formulae was discussed.

All algorithms were implemented in surprisingly few lines of Maude code, illustrating the strength of rewriting for this particular domain. In spite of the reduced size of the code, the implementations seem to be efficient for practical purposes. As a consequence, we have demonstrated how rewriting

can be used not only to experiment with runtime monitoring logics, but also as an implementation language. As an example of future work is the extension of LTL with real-time constraints. Since Maude by itself provides a high-level specification language, one can argue that Maude in its entirety can be used for writing requirements. Further work will show whether this avenue is fruitful. Some of the discussed results and algorithms have been already used in two NASA applications, JPAX and X9, but an extensive experimental assesment of these techniques is left as future work.

Exercises

Exercise 13 *As seen in Chapters 8 and 9, there are different semantics for LTL. This can be quite confusing in practice, because one can write a safety property using LTL thinking of one semantics, say finite-trace semantics, but then use the monitor generation algorithm for the other semantics, say infinite-trace. (1) Give an LTL formula which is valid under one semantics but not under the other. (2) Give an LTL formula whose languages of bad-prefixes are different under the two semantics. (3) Describe an implementation that automatically checks whether a given formula admits different bad prefixes under finite-trace vs. infinite-trace semantics.*

Chapter 9

Efficient Monitoring of ω -Languages

currently from CAV'05 paper [54]; this material will eventually be dissolved in other parts of the book.

Abstract: We present a technique for generating efficient monitors for ω -regular-languages. We show how Büchi automata can be reduced in size and transformed into special, statistically optimal nondeterministic finite state machines, called *binary transition tree finite state machines (BTT-FSMs)*, which recognize precisely the minimal bad prefixes of the original ω -regular-language. The presented technique is implemented as part of a larger monitoring framework and is available for download.

9.1 Introduction

There is increasing recent interest in the area of *runtime verification* [95, 169], which is an area which aims at bridging testing and formal verification. In runtime verification, monitors are generated from system requirements. These monitors observe online executions of programs and check them against requirements. The checks can be either *precise*, with the purpose of detecting existing errors in the observed execution trace, or *predictive*, with the purpose of detecting errors that have not occurred in the observed execution but were “close to happening” and could possibly occur in other executions of the (typically concurrent) system. Runtime verification can be used either during

testing, to catch errors, or during operation, to detect and recover from errors. Since monitoring unavoidably adds runtime overhead to a monitored program, an important technical challenge in runtime verification is that of synthesizing *efficient* monitors from specifications.

Requirements of systems can be expressed in a variety of formalisms, not all of them necessarily easily monitorable. As perhaps best shown by the immense success of programming languages like Perl and Python, regular patterns can be easily devised and understood by ordinary software developers. ω -regular-languages [30, 176] add infinite repetitions to regular languages, thus allowing one to specify properties of reactive systems [130]. The usual acceptance condition in finite state machines (FSM) needs to be modified in order to recognize infinite words, thus leading to Büchi automata [41]. Logics like linear temporal logics (LTL) [130] often provide a more intuitive and compact means to specify system requirements than ω -regular patterns. It is therefore not surprising that a large amount of work has been dedicated to generating (small) Büchi automata from, and verifying programs against, LTL formulae [70, 176, 60, 68].

Based on the belief that ω -languages represent a powerful and convenient formalism to express requirements of systems, we address the problem of generating efficient monitors from ω -languages expressed as Büchi automata. More precisely, we generate monitors that recognize the *minimal bad prefixes* [123] of such languages. A bad prefix is a *finite* sequence of events which cannot be the prefix of any accepting trace. A bad prefix is minimal if it does not contain any other bad prefix. Therefore, our goal is to develop efficient techniques that read events of the monitored program incrementally, and precisely detect when a *bad prefix* has occurred. Dual to the notion of bad prefix is that of a good prefix, meaning that the trace will be accepted for any infinite extension of the prefix.

We present a technique that transforms a Büchi automaton into a special (nondeterministic) finite state machine, called a *binary transition tree finite state machine (BTT-FSM)*, that can be used as a monitor: by maintaining a set of possible states which is updated as events are available. A sequence of events is a bad prefix iff the set of states in the monitor becomes empty. One interesting aspect of the generated monitors is that they may contain a special state, called *neverViolate*, which, once reached, indicates that the specification is *not monitorable* from that moment on. That can mean either that the specification has been fulfilled (e.g., a specification $\diamond(x > 0)$ becomes fulfilled when x is first seen larger than 0), or that from that moment

on, there will always be some possible continuation of the execution trace. For example, the monitor generated for $\Box(a \rightarrow \Diamond b)$ will have exactly one state, *neverViolate*, reflecting the intuition that liveness properties cannot be monitored.

As usual, a program state is abstracted as a set of relevant atomic predicates that hold in that state. However, in the context of monitoring, the evaluation of these atomic predicates can be the most expensive part of the entire monitoring process. One predicate, for example, can say whether the vector $v[1\dots 1000]$ is sorted. Assuming that each atomic predicate has a given evaluation cost and a given probability to hold, which can be estimated apriori either by static or by dynamic analysis, the BTT-FSM generated from a Büchi automaton executes a “conditional program”, called a *binary transition tree (BTT)*, evaluating atomic predicates *by need* in each state in order to statistically optimize the decision to which states to transit. One such BTT is shown in Fig. 6.4.

The work presented in this paper is part of a larger project focusing on *monitoring-oriented programming (MOP)* [36, 35] which is a tool-supported software development framework in which monitoring plays a foundational role. MOP aims at reducing the gap between specification and implementation by integrating the two through monitoring: specifications are checked against implementations at runtime, and recovery code is provided to be executed when specifications are violated. MOP is specification-formalism-independent: one can add one’s favorite or domain-specific requirements formalism via a generic notion of *logic plug-in*, which encapsulates a formal logical syntax plus a corresponding monitor synthesis algorithm. The work presented in this paper is implemented and provided as part of the LTL logic plugin in our MOP framework. It is also available for online evaluation and download on the MOP website [1].

Some Background and Related Work. Automata theoretic model-checking is a major application of Büchi automata. Many model-checkers, including most notably SPIN [100], use this technique. So a significant effort has been put into the construction of small Büchi automata from LTL formulae. Gerth *et al.* [70] show a tableaux procedure to generate on-the-fly Büchi automata of size $2^{O(|\varphi|)}$ from LTL formulae φ . Kesten *et al.* [113] describe a backtracking algorithm, also based on tableaux, to generate Büchi automata from formulae involving both past and future modalities (PTL), but no complexity results are shown. It is known that LTL model-checking is PSPACE-complete [168] and PTL is as expressive and as hard as LTL [131],

though exponentially more succinct [131]. Recently, Gastin and Oddoux [68] showed a procedure to generate standard Büchi automata of size $2^{O(|\varphi|)}$ from PTL via alternating automata. Several works [60, 70] describe simplifications to reduce the size of Büchi automata. Algebraic simplifications can also be applied *a priori* on the LTL formula. For instance, $a \mathcal{U} b \wedge c \mathcal{U} b \equiv (a \wedge c) \mathcal{U} b$ is a valid LTL congruence that will reduce the size of the generated Büchi automaton. All these techniques producing small automata are very useful in our monitoring context because the smaller the original Büchi automaton for the ω -language, the smaller the BTT-FSM. Simplifications of the automaton *with respect to monitoring* are the central subject of this paper.

Kupferman *et al.* [123] classify safety according to the notion of *informativeness*. Informative prefixes are those that “tell the whole story”: they witness the violation (or validation) of a specification. Unfortunately, not all bad prefixes are informative; e.g., the language denoted by $\Box(q \vee \circ(\Box(p))) \wedge \Box(r \vee \circ(\Box(\neg p)))$ does not include any word whose prefix is $\{q, r\}, \{q\}, \{!p\}$. This is a (minimal) bad but not informative prefix, since it does not witness the violation taking place in the next state. One can use the construction described in [123] to build an automaton of size $O(2^{2^{|\varphi|}})$ which recognizes all bad prefixes but, unfortunately, this automaton may be too large to be stored. Our fundamental construction is similar in spirit to theirs but we do not need to apply a subset construction on the input Büchi since we already maintain the set of possible states that the running program can be in. Geilen [69] shows how Büchi automata can be turned into monitors. The construction builds a tableaux similar to [70] in order to produce an FSM of size $O(2^{|\varphi|})$ for recognizing informative good prefixes. Here we detect all the minimal bad prefixes, rather than just the informative ones. Unlike model-checking where a user hopes to see a counter-example that witnesses the violation, when monitoring critical applications one might want to observe a problem as soon as it occurs.

The technique illustrated here is implemented as a plug-in in the MOP *runtime verification (RV)* framework [36, 35]. Other RV tools include JAVA-MAC [115], JPAX [84], JMPAX [165], and EAGLE [18]. JAVA-MAC uses a special interval temporal logic as the specification language, while JPAX and JMPAX support variants of LTL. These systems instrument the JAVA bytecode to emit events to an external monitor observer. JPAX was used to analyze NASA’s K9 Mars Rover code [14]. JMPAX extends JPAX with predictive capabilities. EAGLE is a finite-trace temporal logic and tool for runtime verification, defining a logic similar to the μ -calculus with data-

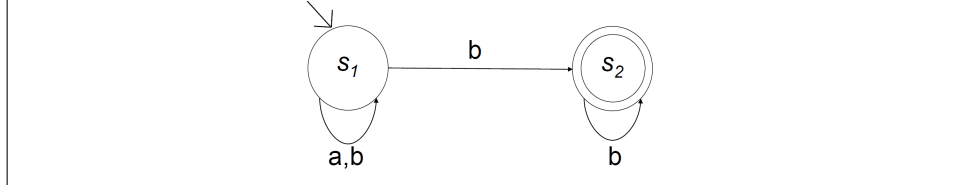


Figure 9.1: Büchi automaton recognizing the ω -regular expression $(a + b)^*b^\omega$

parameterization

9.2 Preliminaries: Büchi Automata

Büchi automata and their ω -languages have been studied extensively during the past decades. They are well suited to program verification because one can check satisfaction of properties represented as Büchi automata statically against transition systems [176, 41]. LTL is an important but proper subset of ω -languages.

Definition 39 A (nondeterministic) standard **Büchi automaton** is a tuple $\langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$, where Σ is an **alphabet**, S is a set of **states**, $\delta: S \times \Sigma \rightarrow 2^S$ is a **transition function**, $S_0 \subseteq S$ is the set of **initial states**, and $\mathcal{F} \subseteq S$ is a set of **accepting states**.

In practice, Σ typically refers to events or actions in a system to be analyzed.

Definition 40 A Büchi automaton $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$ is said to **accept** an infinite word $\tau \in \Sigma^\omega$ iff there is some **accepting run** in the automaton, that is, a map $\rho: \text{Nat} \rightarrow S$ such that $\rho_0 \in S_0$, $\rho_{i+1} \in \delta(\rho_i, \tau_i)$ for all $i \geq 0$, and $\text{inf}(\rho) \cap \mathcal{F} \neq \emptyset$, where $\text{inf}(\rho)$ contains the states occurring infinitely often in ρ . The **language of \mathcal{A}** , $\mathcal{L}(\mathcal{A})$, consists of all words it accepts.

Therefore, ρ can be regarded as an infinite path in the automaton that starts with an initial state and contains at least one accepting state appearing infinitely often in the trace. Fig. 9.1 shows a nondeterministic Büchi automaton for the ω -regular expression $(a + b)^*b^\omega$ that contains all the infinite words over a and b with finitely many a s.

Definition 41 Let $\mathcal{L}(\mathcal{A})$ be the language of a Büchi automaton $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$. A finite word $x \in \Sigma^*$ is a **bad prefix of \mathcal{A}** iff for any $y \in \Sigma^\omega$ the concatenation $xy \notin \mathcal{L}(\mathcal{A})$. A bad prefix is **minimal** if no other bad prefix is a prefix of it.

Therefore, no bad prefix of the language of a Büchi automaton can be extended to an accepted word. Similarly to [41], from now on we may tacitly assume that Σ is defined in terms of propositions over atoms. For instance, the self-transitions of s_1 in Fig. 9.1 can be represented as one self-transition, $a \vee b$.

9.3 Generating a *BTT-FSM* from a Büchi automaton

Not any property can be monitored. For example, in order to check a liveness property one needs to ensure that certain propositions hold infinitely often, which cannot be verified at runtime. This section describes how to transform a Büchi automaton into an efficient *BTT-FSM* that rejects precisely the minimal bad prefixes of the denoted ω -language.

Definition 42 A *monitor FSM (MFSM)* is a tuple $\langle \Sigma, S, \delta, S_0 \rangle$, where $\Sigma = P_A$ is an alphabet, S is a set of states potentially including a special state “*neverViolate*”, $\delta: S \times \Sigma \rightarrow 2^S$ is a transition function with $\delta(\text{neverViolate}, \text{true}) = \{\text{neverViolate}\}$ when $\text{neverViolate} \in S$, and $S_0 \subseteq S$ are initial states.

Note that we take Σ to be P_A , the set of propositions over atoms in A . Like *BTT-FSMs*, *MFSMs* may also have a special *neverViolate* state.

Definition 43 Let $Q_0 \xrightarrow{\theta_1} Q_1 \xrightarrow{\theta_2} \dots \xrightarrow{\theta_j} Q_j$ be a sequence of transitions in the *MFSM* $\langle \Sigma, S, \delta, S_0 \rangle$, generated from $t = \theta_1 \theta_2 \dots \theta_j$, where $Q_0 = S_0$ and $Q_{i+1} = \bigcup_{s \in Q_i} \{\delta(s, \sigma) \mid \theta_{i+1} \models \sigma\}$, for all $0 \leq i < j$. We say that the *MFSM* **rejects** t iff $Q_j = \{\}$.

No finite extension of t will be **rejected** if $\text{neverViolate} \in Q_j$.

From Büchi to *MFSM*. We next describe two simplification procedures on a Büchi automaton that are sound w.r.t. monitoring, followed by the construction of an *MFSM*. The first procedure identifies segments of the automaton which cannot lead to acceptance and can therefore be safely removed. As we will show shortly, this step is necessary in order to guarantee the soundness of the monitoring procedure. The second simplification identifies states with the property that if they are reached then the corresponding requirement cannot be violated by any *finite* extension of the trace, so

monitoring is ineffective from there on. Note that reaching such a state does not necessarily mean that a good prefix has been recognized, but only that the property is *not monitorable* from there on.

Definition 44 Let $\langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$ be a Büchi automaton, C a connected component of its associated graph, and $\text{nodes}(C)$ the states associated to C . We say that C is **isolated** iff for any $s \in \text{nodes}(C)$ and $\sigma \in \Sigma$, it is the case that $\delta(s, \sigma) \subseteq \text{nodes}(C)$. We say that C is **total** iff for any $s \in \text{nodes}(C)$ and event θ , there are transitions σ such that $\theta \models \sigma$ and $\delta(s, \sigma) \cap \text{nodes}(C) \neq \emptyset$.

Therefore, there is no way to escape from an isolated connected component, and regardless of the upcoming event, it is always possible to transit from any node of a total connected component to another node in that component.

Removing Bad States. The next procedure removes states of the Büchi automaton which cannot be part of any accepting run (see Definition 2). Note that any state appearing in such an accepting run must eventually *reach* an accepting state. This procedure is fundamentally inspired by strongly-connected-component-analysis [113, 176], used to check emptiness of the language denoted by a Büchi automaton. Given a Büchi automaton $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$, let $U \subseteq S$ be the largest set of states such that the language of $\langle \Sigma, S, \delta, U, \mathcal{F} \rangle$ is empty. The states in U are unnecessary in \mathcal{A} , because they cannot change its language. Fortunately, U can be calculated effectively as the set of states that *cannot reach* any cycle in the graph associated to \mathcal{A} which contains at least one accepting state in \mathcal{F} . Fig. 9.2 shows an algorithm to do this.

<p>INPUT : A Büchi automaton \mathcal{A} OUTPUT : A smaller Büchi automaton \mathcal{A}' such that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$. REMOVE_BAD_STATES : for each maximal connected component C of \mathcal{A} if $(C$ is isolated and $\text{nodes}(C) \cap \mathcal{F} = \emptyset)$ then mark all states in C “bad” DFS_MARK_BAD ; REMOVE_BAD</p>

Figure 9.2: Removing bad states

The loop identifies maximal isolated connected components which do not contain any accepting states. The nodes in these components are marked as “bad”. The procedure DFS_MARK_BAD performs a depth-first-search in the

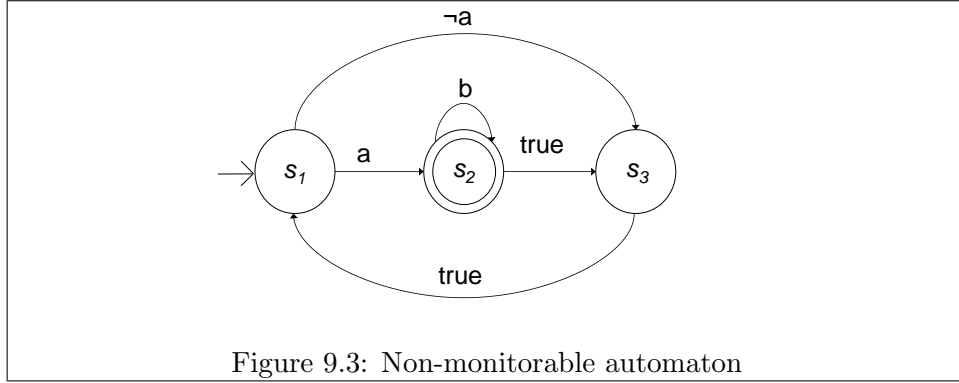
graph and marks nodes as “bad” when all outgoing edges lead to a “bad” node. Finally, the procedure `REMOVE_BAD` removes all the bad states. The runtime complexity of this algorithm is dominated by the computation of maximal connected components. In our implementation, we used Tarjan’s $O(V + E)$ double DFS [41]. The proof of correctness is simple and it appears in Appendix A. The Büchi automaton \mathcal{A}' produced by the algorithm in Fig. 9.2 has the property that there is some proper path from any of its states to some accepting state. One can readily generate an *MFSM* from a Büchi automaton \mathcal{A} by first applying the procedure `REMOVE_BAD_STATES` in Fig. 9.2, and then ignoring the acceptance conditions.

Theorem 18 *The *MFSM* generated from a Büchi automaton \mathcal{A} as above rejects precisely the minimal bad prefixes of $\mathcal{L}(\mathcal{A})$.*

Proof: Let $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$ be the original Büchi automaton, let $\mathcal{A}' = \langle \Sigma, S', \delta', S'_0, \mathcal{F} \rangle$ be the Büchi automaton obtained from \mathcal{A} by applying the algorithm in Fig. 9.2, and let $\langle \Sigma, S', \delta', S'_0 \rangle$ be the corresponding *MFSM* of \mathcal{A}' . For any finite trace $t = \theta_1 \dots \theta_j$, let us consider its corresponding sequence of transitions in the *MFSM* $Q_0 \xrightarrow{\theta_1} \dots \xrightarrow{\theta_j} Q_j$, where Q_0 is S'_0 . Note that the trace t can also be regarded as a sequence of letters in the alphabet Σ of \mathcal{A} , because we assumed Σ is P_A and because there is a bijection between propositions in P_A and A -events. All we need to show is that t is a bad prefix of \mathcal{A}' if and only if $Q_j = \emptyset$. Recall that \mathcal{A}' has the property that there is some non-empty path from any of its states to some accepting state. Thus, one can build an infinite path in \mathcal{A}' starting with any of its nodes, with the property that some accepting state occurs infinitely often. In other words, Q_j is not empty iff the finite trace t is the prefix of some infinite trace in $\mathcal{L}(\mathcal{A}')$. This is equivalent to saying that Q_j is empty iff the trace t is a bad prefix in \mathcal{A}' . Since Q_j empty implies $Q_{j'}$ empty for any $j > j'$, it follows that the *MFSM* rejects precisely the minimal bad prefixes of \mathcal{A} . \square

Theorem 1 says that the *MFSM* obtained from a Büchi automaton as above can be used as a monitor for the corresponding ω -language. Indeed, one only needs to maintain a current set of states Q , initially S'_0 , and transform it accordingly as new events θ are generated by the observed program: if $Q \xrightarrow{\theta} Q'$ then set Q to Q' ; if Q ever becomes empty then report violation. Theorem 1 tells us that a violation will be reported as soon as a bad prefix is encountered.

Collapsing Never-Violate States. Reducing runtime overhead is crucial in runtime verification. There are many situations when the monitoring



process can be safely stopped, because the observed finite trace cannot be finitely extended to any bad prefix. The following procedure identifies states in a Büchi automaton which cannot lead to the violation of any *finite* computation. For instance, the Büchi automaton in Fig. 9.3 can only reject infinite words in which the state s_2 occurs finitely many times; moreover, at least one transition is possible at any moment. Therefore, the associated *MFSM* will never report a violation, even though there are infinite words that are not accepted. We call such an automaton *non-monitorable*. This example makes it clear that if a state like s_1 is ever reached by the monitor, it does *not* mean that we found a good prefix, but that we could *stop looking* for bad prefixes.

Let $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$ be a Büchi automaton *simplified with REMOVE_BAD_STATES*. The procedure in Fig. 9.4 finds states which, if reached by a monitor, then the monitor can no longer detect violations regardless of what events will be observed in the future. The procedure first identifies the total connected components. According to the definition of totality, once a monitor reaches a state of a total connected component, the monitor will have the possibility to always transit within that connected component, thus never getting a chance to report violation. All states of a total component can therefore be marked as “never violate”. Other states can also be marked as such if, for any events, it is possible to transit from them to states already marked “never violate”; that is the reason for the disjunction in the second conditional. The procedure finds such nodes in a depth-first-search. Finally, **COLLAPSE-NEVER_VIOLATE** collapses all components marked “never violate”, if any, to a distinguished node, *neverViolate*, having just a *true* transition to itself. If any collapsed node was in the initial set of states, then the entire automaton is collapsed to *neverViolate*. The procedure **GENERATE_MFSM** produces an *MFSM* by ignoring accepting conditions. Taking as input this *MFSM*, say $\langle \Sigma, S', \delta', S'_0 \rangle$, cost function ς , and probability

```

INPUT : A Büchi automaton  $\mathcal{A}$ , cost function  $\varsigma$ , and probability function  $\pi$ .
OUTPUT : An effective BTT-FSM monitor rejecting the bad prefixes of  $\mathcal{L}(\mathcal{A})$ .
COLLAPSE_NEVER_VIOLATE :
  for each maximal connected component  $C$  of  $\mathcal{A}$ 
    if (  $C$  is total ) then mark all states in  $C$  as “never violate”
  for each  $s$  in depth-first-search visit
    if (  $\bigvee \{\sigma \mid \delta(s, \sigma) \text{ contains some state marked “never violate”} \}$  )
      then mark  $s$  as “never violate”
  COLLAPSE_NEVER_VIOLATE ; GENERATE_MFSM ; GENERATE_BTT-FSM

```

Figure 9.4: Collapsing non-monitorable states

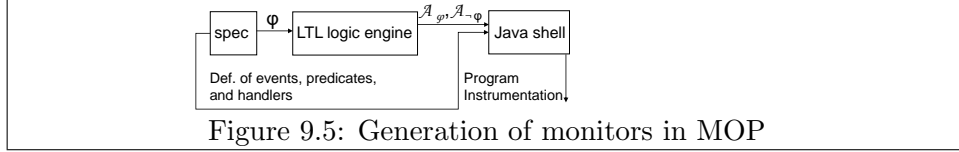


Figure 9.5: Generation of monitors in MOP

function π , **GENERATE.BTT-FSM** constructs a *BTT-FSM* $\langle A, S', btt, S'_0 \rangle$, where A corresponds to the set of atoms from which the alphabet Σ is built, and the map *btt*, here represented by a set of pairs, is defined as follows:

$$\begin{aligned}
 btt = & \{ (neverViolate, \{neverViolate\}) \mid neverViolate \in S' \} \cup \\
 & \{ (s, \beta_s) \mid s \in S' - \{neverViolate\} \wedge \beta_s \models \mu_s \}, \text{ where} \\
 & \beta_s \text{ optimally implements } \mu_s \text{ w.r.t. } \varsigma \text{ and } \pi, \text{ with } \mu_s = \oplus (\bigcup \{ [\sigma : s'] \mid s' \in \delta'(s, \sigma) \})
 \end{aligned}$$

The symbol \oplus denotes concatenation on a set of multi-transitions. Optimal *BTTs* β_s are generated like in Section 6.2. Proof of correctness appears in Appendix A.

9.4 Monitor Generation and MOP

We have shown that one can generate from a Büchi automaton a *BTT-FSM* recognizing precisely its bad prefixes. However, it is still necessary to integrate the *BTT-FSM* monitor within the program to be observed. *Monitoring-oriented programming* (MOP) [35] aims at merging specification and implementation through generation of runtime monitors from specifications and integration of those within implementation. In MOP, the task of generating monitors is divided into defining a logic engine and a language shell. The logic engine is concerned with the translation of specifications given as logical formulae into monitoring (pseudo-)code. The shell is responsible for the integration of the monitor within the application.

Fig. 9.5 captures the essence of the synthesis process of LTL monitors in MOP using the technique described in this paper. The user defines specifications either as annotations in the code or in a separate file. The specification contains definitions of events and state predicates, as well as LTL formulae expressing trace requirements. These formulae treat events and predicates as atomic propositions. Handlers are defined to track violation or validation of requirements. For instance, assume the events a and b denote the login and the logoff of the same user, respectively. Then the formula $\Box(a \rightarrow \circ(\neg a \mathcal{U} b))$ states that the user cannot be logged in more than once. A violation handler could be declared to track the user who logged in twice. The logic engine is responsible for the translation of the formulae φ and $\neg\varphi$ into two *BTT-FSM* monitors. One detects violation and the other validation of φ . Note that if the user is just interested in validation (no violation handler), then only the automaton for negation is generated. Finally, the language shell reads the definition of events and instruments the code so that the monitor will receive the expected notifications.

We used LTL2BA [140] to generate standard Büchi automata from LTL formulae. The described procedures are implemented in JAVA. This software and a WWW demo are available from the MOP website [1].

9.4.1 Evaluation

Table 1 shows *BTT-FSM* monitors for some LTL formulae. The *BTT* definition corresponding to a state follows the arrow (\rightsquigarrow). Initial states appear in brackets. For producing this table, we used the same cost and probabilities for all events and selected the smallest *BTT*. The first formula cannot be validated by monitoring and presents the permanent possibility to be violated; that is why its *BTT-FSM* does not have a *neverViolate* state. The second formula can never be violated since event a followed by event b can always occur in the future, so its *BTT-FSM* consists of just one state *neverViolate*. The last formula shows that our procedure does not aim at distinguishing validating from non-violating prefixes.

Table 2 shows that our technique can not only identify non-monitorable formulae, but also reduce the cost of monitoring by collapsing large parts of the Büchi automaton. We use the symbols \heartsuit , \clubsuit , and \spadesuit to denote, respectively, the effectiveness of REMOVE_BAD_STATES, the first, and the second loop of COLLAPSE_NEVER_VIOLATE. The first group contains non-monitorable formulae. The next contains formulae where monitor size could not be reduced by our procedures. The third group shows formulae where our

Temporal Formula	<i>BTT-FSM</i>
$\Box(a \rightarrow b \mathcal{U} c)$	$[s_0] \rightsquigarrow c ? (b ? s_0 s_1 : s_0) : (a ? (b ? s_1 : \emptyset) : (b ? s_0 s_1 : s_0))$ $s_1 \rightsquigarrow b ? (c ? s_0 s_1 : s_1) : (c ? s_0 : \emptyset)$
$\Box(a \rightarrow \Diamond b)$	$[neverViolate] \rightsquigarrow \{neverViolate\}$
$a \mathcal{U} b \mathcal{U} c$	$[s_0] \rightsquigarrow c ? neverViolate : (a ? (b ? s_0 s_1 : s_0) : (b ? s_1 : \emptyset))$ $s_1 \rightsquigarrow c ? neverViolate : (b ? s_1 : \emptyset)$ $neverViolate \rightsquigarrow \{neverViolate\}$

Table 9.1: *BTT-FSMs* generated from temporal formulae

simplifications could significantly reduce the monitor size. The last group shows examples of “accidentally” safe and “pathologically” safe formulae from [123]. A formula φ is accidentally safe iff not all bad prefixes are “informative” [123] (i.e., can serve as a witness for violation) but all computations that violate φ have an informative bad prefix. A formula φ is pathologically safe if there is a computation that violates φ and has no informative bad prefix. Since we detect *all* minimal bad prefixes, informativeness does not play any role in our approach. Both formulae are monitorable. For the last formula, in particular, a minimal bad prefix will be detected as soon as the monitor observes a $\neg a$, having previously observed a $\neg b$. One can generate and visualize the *BTT-FSMs* of all these formulae, and many others, online at [1].

9.5 Conclusions

Not all properties a Büchi automaton can express are monitorable. This paper describes transformations that can be applied to extract the monitorable components of Büchi automata, reducing their size and the cost of runtime verification. The resulting automata are called *monitor finite state machines* (*MFSMs*). The presented algorithms have polynomial running time in the size of the original Büchi automata and have already been implemented. Another contribution of this paper is the definition and use of *binary transition trees* (*BTTs*) and corresponding finite state machines (*BTT-FSMs*), as well as a translation from *MFSMs* to *BTT-FSMs*. These special-purpose state machines encode optimal evaluation paths of boolean propositions in transitions.

We used LTL2BA [140] to generate Büchi automata from LTL, and












Temporal Formula	# states	# transitions	symlif.
$\diamond a$	2 , 1	3 , 1	
$a \mathcal{U} \circ(\diamond b)$	3 , 1	5 , 1	
$\Box(a \wedge b \rightarrow \diamond c)$	2 , 1	4 , 1	
$a \mathcal{U} (b \mathcal{U} (c \mathcal{U} (\diamond d)))$	2 , 1	3 , 1	
$a \mathcal{U} (b \mathcal{U} (c \mathcal{U} \Box(d \rightarrow \diamond e)))$	5 , 1	15 , 1	
$\neg a \mathcal{U} (b \mathcal{U} (c \mathcal{U} \Box(d \rightarrow \diamond e)))$	12 , 1	51 , 1	
$\neg \diamond a$	1 , 1	1 , 1	
$\Box(a \rightarrow b \mathcal{U} c)$	2 , 2	4 , 4	
$a \mathcal{U} (b \mathcal{U} (c \mathcal{U} d))$	4 , 4	10 , 10	
$a \wedge \circ(\diamond b) \wedge \diamond(\Box(e))$	5 , 4	11 , 6	
$a \wedge \circ(\diamond b) \wedge \circ(\diamond c) \wedge \diamond(\Box(e))$	9 , 6	29 , 12	
$a \wedge \circ(\diamond b) \wedge \circ(\diamond c) \wedge \circ(\diamond d) \wedge \diamond(\Box(e))$	17 , 10	83 , 30	
$a \wedge \circ(\neg(\Box(b \rightarrow c \mathcal{U} d))) \wedge \diamond(\Box(e))$	7 , 5	20 , 10	
$\Box(a \vee \circ(\Box(c))) \wedge \Box(b \vee \circ(\Box(\neg c)))$	3 , 3	5 , 5	
$(\Box(a \vee \diamond(\Box(c))) \wedge \Box(b \vee \diamond(\Box(\neg c)))) \vee \Box(a) \vee \Box(b)$	12 , 6	43 , 22	

Table 9.2: Number of states and transitions before and after monitoring simplifications

JAVA to implement the presented algorithms. Our algorithms, as well as a graphical HTML interface, are available at [1]. This work is motivated by, and is part of, a larger project aiming at promoting monitoring as a foundational principle in software development, called *monitoring-oriented programming* (MOP). In MOP, the user specifies formulae, atoms, cost and probabilities associated to atoms, as well as violation and validation handlers. Then all these are used to automatically generate monitors and integrate them within the application.

This work is concerned with monitoring *violations* of requirements. In the particular case of LTL, *validations* of formulae can also be checked using the same technique by monitoring the negation of the input formula. Further work includes implementing the algorithm defined in [68] for generating Büchi automata of size $2^{O(|\varphi|)}$ from PTL, combining multiple formulae in a single automaton as showed by Ezick [61] so as to reduce redundancy of proposition evaluations, and applying further (standard) NFA simplifications to *MFSM*.

Appendix A. Proof of Correctness

(*Correctness of REMOVE_BAD_STATES*) Let $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$ be the input and $\mathcal{A}' = \langle \Sigma, S', \delta', S'_0, \mathcal{F} \rangle$ the output Büchi automata of the procedure. We want to show that their denoted languages are the same. Let ρ be an accepting run of \mathcal{A} . Then ρ can be fragmented in $\rho' \rho''$ where ρ' is the prefix whose states appear only finitely many times in ρ . Each state in ρ'' is therefore reachable from any other and therefore must be in a connected component where some states are in \mathcal{F} . The converse is also true, i.e., any connected component reachable from the set of initial states having at least one state in \mathcal{F} generates accepting runs (from [41] pp. 129). It is then immediate to notice that no accepting run ρ contains states that can never reach a state in \mathcal{F} . It turns out that any state belonging to an isolated component with no accepting states can be trivially removed as well as any state that can *only* make transitions to others which never reach an accepting state. This reachability analysis is performed in a depth-first-search as usual. Since only states that will never appear in accepting runs of \mathcal{A} are removed, it follows that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

(*Correctness of COLLAPSE_NEVER_VIOLATE*) First, we need to show that the *MFSM* produced by `GENERATE_MFSM` still detects bad prefixes of $\mathcal{L}(\mathcal{A})$, where \mathcal{A} is the input Büchi automaton having the property that all states can reach accepting states. From the observation that all states that have been collapsed to *neverViolate* can reach an accepting state, the proof of the first part is similar to that of Theorem 1 and is omitted. So, the *MFSM* is still a monitor for the bad prefixes of $\mathcal{L}(\mathcal{A})$. In other words, this simplification is sound w.r.t. monitoring minimal bad prefixes. In addition to this, we need to show that the state *neverViolate* is reached *only if* violations of the requirement can no longer occur. We prove this second correctness criteria by a case analysis on the shape of the *MFSM* associated to the input Büchi. Each case corresponds to a loop in `COLLAPSE_NEVER_VIOLATE`.

(case 1) Let $Q_0 \xrightarrow{\theta_1} Q_1 \xrightarrow{\theta_2} \dots \xrightarrow{\theta_i} Q_j$ be a sequence of transitions on the trace $\theta_1 \theta_2 \dots \theta_j$ produced by the *MFSM* corresponding to the input Büchi automaton \mathcal{A} . Recall that we assume `REMOVE_BAD_STATES` is called before `COLLAPSE_NEVER_VIOLATE`. That is, all states in the input automaton reach some state in \mathcal{F} . If some node in a total connected component C of the

graph associated to \mathcal{A} belongs to Q_j then it is not possible for any finite trace with prefix $\theta_1\theta_2...\theta_j$ to be rejected by the corresponding *MFSM* since it is always possible to make a transition between nodes in C from the definition of totality. Those states in C can be marked as “never violate” meaning that they denote the special *neverViolate* state.

(case 2) Let $Q_0 \xrightarrow{\theta_1} Q_1 \xrightarrow{\theta_2} \dots \xrightarrow{\theta_j} Q_j$ be a sequence of transitions produced on the trace $\theta_1\theta_2...\theta_j$ by the *MFSM* defined as before. If any state belongs to Q_j with the property that for further events θ_{j+1} there exists a transition to a state marked “never violate”, then there must exist some state marked “never violate” in Q_{j+1} . Such states can be found by checking the validity of the disjunction of propositions labeling the edges of transitions to states marked “never violate”. Since the monitor will definitely reach a state marked “never violate” in the trace $\theta_1\theta_2...\theta_j\theta_{j+1}$, it is therefore safe to reach “never violate” in Q_j as well, since violations are impossible from event j on.

The states marked with the “never violate” label can therefore be collapsed, since a violation cannot occur if any of these states is reached. The *BTT-FSM* is generated from the *MFSM* according to the defined construction.

Appendix B. Complexity Results

We use the definition of Binary Decision Trees (BDTs) due to Garey [67]. As we will show next, Garey’s BDTs serve as a procedure to identify one among a set of possible objects characterizing some aspect of interest. Hyafil and Rivest [107] proved the Optimal BDT problem NP-complete. Moret [138] shows that these BDTs are a restricted form of decision trees similar to binary transition trees as defined here.

Let $X = \{x_1, \dots, x_n\}$ be a finite set of objects and $\mathcal{T} = \{T_1, \dots, T_t\}$ a finite set of tests. For each test T_i , $1 \leq i \leq t$, and object x_j , $1 \leq j \leq n$, we either have $T_i(x_j) = \text{true}$ or $T_i(x_j) = \text{false}$. A binary decision tree associates tests in the root and all other internal nodes, and associates objects in X to terminal nodes. The *optimal decision tree* problem is to determine whether there exists a decision tree with cost less than or equal to w which completely identifies each element in X , given \mathcal{T} and X . The cost of this tree is defined as $\sum_{x_i \in X} p(x_i)$, where $p(x_i)$ is the length of the path from the root to the terminal identifying x_i .

One might think of the objects in X as possible answers to a question. The tests in \mathcal{T} serve to prune the data set of answers. A decision tree defines

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	
T_1	T	F	T	F	T	F	F	T_1
T_2	F	T	F	F	F	T	T	T_6
T_3	T	F	F	F	F	F	F	T_3
T_4	F	F	F	F	T	F	F	T_4
T_5	F	F	F	F	F	T	F	T_5
T_6	F	F	F	F	F	F	T	T_2

Figure 9.6: Table of tests and a possible BDT for this table

possible sequences of questions to ask in order to give a unique answer among those in X . The table of Fig. 9.6 appears in [67] and denotes the set of tests available to use in some binary decision tree.

The definition above [67] and the NP-completeness proof for the Optimal BDT problem [107] assume the table provided as input is unambiguous and completely identifies the objects in X . That is, any object in X can be distinguished by applying some sequence of tests. The decision tree in Fig. 9.6 appears in [67] as an identification procedure for the tests and objects defined in the adjacent table. The subtree to the left of a node denotes objects whose answers to the test labeling that node are *true* and the result of all other tests applied in the path to the root are consistent. The right subtree is similar. This is similar to binary transition trees. However, it is worth noting that in order to identify x_j one does not need to test all T_i having $T_i(x_j) = \text{true}$. Observe this in particular for x_6 and x_7 . It means that decision trees prune the data set in each test they make. A decision can be made whenever it is possible to identify an object. For instance, no object but x_7 assigns *false* to T_1 and *true* to T_6 . So this is a sufficient condition to identify x_7 . Moret showed that the construction of optimal binary decision trees, similar to *BTTs* (where tests take the form of boolean proposition over a set of atoms) is an NP-hard [138] problem. That implies that our *BTT* problem is also NP-hard.

Exercises

Exercise 14 Consider the following three (infinite-trace) LTL formulae from Table 9.2: $\Box(a \wedge b \rightarrow \Diamond c)$, $\Box(a \rightarrow b \mathcal{U} c)$, and, respectively, $a \wedge \Diamond(\Diamond b) \wedge \Diamond(\Box(e))$. For each of them do the following: (1) give a Buchi automaton that has the same language; (2) give an MFSM corresponding to (1) that rejects precisely the minimal bad prefixes (see Theorem 18); (3) optimize the MFSM in (2) by collapsing the non-monitorable states (algorithm in Figure 9.4); (4) give the optimal BTT-FSM corresponding to (3).

Chapter 10

Efficient Monitoring of “Always Past” Temporal Safety

This chapter contains material from [97]. See also [96].

Abstract: The problem of testing whether a finite execution trace of events generated by an executing program violates a linear temporal logic (LTL) formula occurs naturally in runtime analysis of software. Two efficient algorithms for this problem are presented in this paper, both for checking safety formulae of the form “always P ”, where P is a past time LTL formula. The first algorithm is implemented by rewriting and the second synthesizes efficient code from formulae. Further optimizations of the second algorithm are suggested, reducing space and time consumption. Special operators suitable for writing succinct specifications are discussed and shown equivalent to the standard past time operators. This work is part of NASA’s PathExplorer project, the objective of which is to construct a flexible framework for efficient monitoring and analysis of program executions.

Like in future time LTL, but dually, we can have various ways to interpret $\circ\varphi$. The stationary interpretation appears to be different from the strong / weak interpretation. That is, strong reduces to weak and viceversa, but it is not clear how stationary reduces to any of these or to anything else.

10.1 Introduction

The work presented in this paper is part of a project at NASA Ames Research Center, called PathExplorer [85, 84, 90, 83, 146], that aims at developing a practical testing environment for NASA software developers. The basic idea of the project is to analyze the execution trace of a running program to detect errors. The errors that we are considering at this stage are multi-threading errors such as deadlocks and data races, and non-conformance with linear temporal logic specifications, which is the main focus of this paper.

Linear Temporal Logic (LTL) [143, 128, 130] is a logic for specifying properties of reactive and concurrent systems. The models of LTL are infinite execution traces, reflecting the behavior of such systems as ideally always being ready to respond to requests, operating systems being a typical example. LTL has been mainly used to specify properties of concurrent and interactive down-scaled models of real systems, so that fully formal correctness proofs could subsequently be carried out, for example using theorem provers or model checkers (see for example [101, 98, 91]). However, formal proof techniques are usually not scalable to real sized systems without a substantial effort to abstract the system more or less manually to a model which can be analyzed. Model checking of programs has received an increased attention from the formal methods community within the last couple of years, and several systems have emerged that can directly model check source code, such as Java and C [92, 175, 56, 102, 51, 17, 142]. Stateless model checkers [72, 172] try to avoid the abstraction process by not storing states. Although these systems provide high confidence, they scale less well because most of their internal algorithms are exponential or worse.

Testing scales well, and is by far the most used technique in practice to validate software systems. The merge of testing and temporal logic specification is an attempt to achieve the benefits of both approaches, while avoiding some of the pitfalls of ad hoc testing and the complexity of full-blown theorem proving and model checking. Of course, there is a price to pay in order to obtain a scalable technique: the loss of coverage. The suggested

framework can only be used to examine single execution traces, and can therefore not be used to prove a system correct. Our work is based on the belief that software engineers are willing to trade coverage for scalability, so our goal is to provide tools that are completely automatic, implement very efficient algorithms and find *many* errors in programs. A longer term goal is to explore the use of conformance with a formal specification to achieve fault tolerance. The idea is that the failure may trigger a recovery action in the monitored program.

The idea of using LTL in program testing is not new. It has already been pursued in commercial tools such as Temporal Rover (TR) [57], which stimulated our work in a major way. In TR, future and past time LTL properties are stated as formal comments within the program at chosen program points, like assertions. These formal comments are then, by a pre-processor, translated into code, which is inserted at the position of the comments, and executed whenever reached during program execution¹. The MaC tool [127] is another example of a runtime monitoring tool that has inspired this work. Here Java bytecode is automatically instrumented to generate events of interest during the execution. Of special interest is the temporal logic used in MaC, which can be classified as a past time interval logic convenient for expressing monitoring properties in a succinct way. A theoretical contribution in this paper is Theorem 19, which shows that the MaC temporal logic, together with 10 others, is equivalent to the standard past time temporal logic. The path exploration tool described in [79] uses a future time temporal logic formula to guide the execution of a program for debugging purposes. Hence, the role of a temporal logic formula is turned around from monitoring a trace to generation of a trace.

Past time LTL has been shown to have the same expressiveness as future time LTL [65]. However, past time LTL is exponentially more succinct than future time LTL [131]. For example, a property like *"every response should be preceded by a request"* can be easily stated in past time logic (reflecting directly the previous sentence), but the corresponding future time representation becomes *"it's not the case that (there is no request until (there is a response and no request))"*. Hence, past time LTL is more convenient for specifying certain properties, and is the focus of this paper.

We present two efficient monitoring algorithms for checking safety formulae of the form "always P ", where P is a past time LTL formula, one based on formula rewriting and one based on synthesizing efficient monitoring code

¹The implementation details of TR are not public.

from a formula. The rewriting-based algorithm illustrates how rewriting can be used to easily and elegantly define new logics for monitoring. This may be of interest when experimenting with logics, or if logics are domain specific and change with the application, or if one simply wants a small and elegant implementation. The synthesis-based algorithm, on the other hand, generates a very effective monitor for the particular past time logic, and focuses on efficiency. It is also better suited for generating code that can be inserted in the monitored program, in contrast to the rewriting approach, where a rewriting engine must be called by an external call.

The first algorithm is implemented by rewriting using Maude [45, 47, 48], an executable specification language whose main operational engine is based on term rewriting. Since flexibility with respect to defining/modifying monitoring logics is a very important factor at this stage in the development of PathExplorer, we have actually developed a general framework using Maude which allows one to easily and effectively define new logics for runtime analysis and to monitor execution traces against formulae in these logics. The rewriting algorithm presented in this paper instantiate that framework to our logic of interest, past time LTL. The second algorithm presented in this paper is designed to be as efficient and specialized as possible, thus adding the minimum possible amount of runtime overhead. It essentially synthesizes a special purpose, efficient monitoring code from formulae, which is further compiled into an executable monitor. Further optimizations of the second algorithm are suggested, making each monitoring step typically run in time lower than the size of the monitored formula. Both algorithms are based on the fact that the semantics of past time LTL can be defined recursively in such a way that one only needs to look one step, or event, backwards in order to compute the new truth value of a formula and of its subformulae, thus allowing one to process and then discard the events as they are received from the instrumented program. Several special operators suitable for writing succinct monitoring safety specifications are introduced and shown semantically equivalent to the standard past time operators.

Section 10.2 gives a short description of the PathExplorer architecture, putting the presented work in context. Section 10.3 recalls past time LTL and introduces several monitoring operators together with their semantics, then discusses several past time logics and finally shows their equivalences. Section 10.4 first presents our rewriting-based framework for defining and executing new monitoring logics, and then shows how past time LTL fits into this framework. Section 10.5 finally explains our monitor-synthesis algorithm,

Figure 10.1: Overview of the PaX observer.

together with optimizations and two ways to implement it. Section 10.6 concludes the paper.

10.2 The PathExplorer Architecture

PathExplorer, PaX, is a flexible environment for monitoring and analyzing program executions. A program (or a set of programs) to be monitored, is supposed to be instrumented to emit execution events to an observer, which then examines the events and checks whether they satisfy certain user-given constraints. We first give an overview of the *observer* that monitors the event stream. Then we discuss how a program is instrumented for monitoring of temporal logic properties. The instrumentation presented is specialized to Java, but the principles carry over to any programming language.

10.2.1 The Observer

The constraints to be monitored can be of different kinds and defined in different languages. Each kind of constraint is represented by a module. Such a constraint module in principle implements a particular logic or program analysis algorithm. Currently there are modules for checking deadlock potentials, data race potentials, and for checking temporal logic formulae in different logics. Amongst the latter, several modules have been implemented for checking future time temporal logic, and the work presented in this paper is the basis for a module for checking past time logic formulae. In general, the user can program new constraint modules and in this manner extend PaX in an easy way.

The system is defined in a component-based way, based on a dataflow view, where components are put together using a “pipeline” operator, see Figure 10.1. The dataflow between any two components is a stream of events in simple text format, without any a-priori assumptions about the format of the events; the receiving component just ignores events it cannot recognize. This simplifies composition and allows for components to be written in

different languages and in particular to define observers of arbitrary systems, programmed in a variety of programming languages. This latter fact is important at NASA since several systems are written in a mixture of C, C++ and Java.

The central component of the PaX system is a so-called *dispatcher*. The dispatcher receives events from the executing program or system and then retransmits the event stream to each of the constraint modules. Each module is running in its own process with one input pipe, only dealing with events that are relevant to the module. For this purpose each module is equipped with an event parser. The dispatcher takes as input a configuration script, which specifies a list of commands - a command for each module that starts the module in a process. The dispatcher may read its input event stream from a file, or alternatively from a socket, to which the instrumented running program must write the event stream. In the latter case, monitoring can happen on-the-fly as the event stream is produced, and potentially on a different computer than the observed system.

10.2.2 Code Instrumentation

The program or system to be observed must be instrumented to emit execution events to the dispatcher (writing them to a file or to a socket as discussed above). We have currently implemented an automated instrumentation package for Java bytecode using the Java bytecode engineering tool JTrek [112]. The instrumentation package together with PathExplorer is called Java PathExplorer (JPaX). Given information about what kind of events to be emitted, the instrumentation package instruments the bytecode by inserting extra code for emitting events. For deadlock analysis, for example, events are generated that inform about lock acquisitions and releases. For temporal logic monitoring, one specifies the variables to be observed, and what predicates over these variables one wants to refer to in the temporal properties to be monitored. Imagine for example that the observer monitors the formula: “*always p*”, involving the predicate p , and that p is intended to be defined as $p \equiv x > y$, where x and y are static variables defined in a class C . In this case all updates to these variables must be instrumented, such that an update to any of them causes the predicate to be evaluated, and a toggle p to be emitted to the observer in case it has changed. The instrumentation script is written in Java (using reflection), but in essence can be represented as follows:

Figure 10.2: Events corresponding to observing predicate $p \equiv x > y$.

```
monitor C.x, C.y;  
proposition p is C.x > C.y;
```

The code will then be instrumented to emit changes in the predicate p . More specifically, first the initial value of the predicate is transmitted to the observer. Subsequently, whenever one of the two variables is updated, the predicate is evaluated, and in case its value has changed since last evaluation, the predicate name p is transmitted to the observer as a toggle. The observer keeps track of the value of the predicate, based on its initial value, and the subsequent predicate toggles. Figure 10.2 shows an execution trace where x and y initially are 0, and then subsequently updated. The corresponding values of p are shown. Also shown are the events that are sent to the observer. That is, the initial value of p and the subsequent p toggles.

10.3 Finite Trace Past Time LTL

In this section we remind some basic notions of finite trace linear past time temporal logic [128, 130], establish some conventions and introduce some operators that we found particularly useful for runtime monitoring. We emphasize that the semantics of past time LTL can be elegantly defined recursively, thus allowing us to implement monitoring algorithms that only need to look one step backwards. We also show that past time LTL can be entirely defined using just the special operators, that were introduced essentially because of practical needs, thus strengthening our belief that past time LTL is an appropriate candidate logic for expressing monitoring safety requirements.

10.3.1 Syntax

We allow the following constructors for formulae, where A is a finite set of “atomic propositions”:

$$\begin{aligned}
 F ::= & \text{ true } \mid \text{ false } \mid A \mid \neg F \mid F \text{ op } F \\
 & \text{(propositional operators)} \\
 & \odot F \mid \Diamond F \mid \Box F \mid F \mathcal{S}_s F \mid F \mathcal{S}_w F \\
 & \text{(standard past time operators)} \\
 & \uparrow F \mid \downarrow F \mid [F, F)_s \mid [F, F)_w \\
 & \text{(monitoring operators)}
 \end{aligned}$$

The propositional binary operators, op , are the standard ones, that is, disjunction, conjunction, implication, equivalence, and exclusive disjunction.

The standard past time and the monitoring operators are often called “temporal operators”, because they refer to other (past) moments in time. The operator $\odot F$ should be read “previously F ”; its intuition is that F held at the immediately previous moment in time. $\Diamond F$ should be read “eventually in the past F ”, with the intuition that there is some past moment in time when F was true. $\Box F$ should be read “always in the past F ”, with the obvious meaning. The operator $F_1 \mathcal{S}_s F_2$, which should be read “ F_1 strong since F_2 ”, reflects the intuition that F_2 held at some moment in the past and, since then, F_1 held all the time. $F_1 \mathcal{S}_w F_2$ is a weak version of “since”, read “ F_1 weak since F_2 ”, saying that either F_1 was true all the time or otherwise $F_1 \mathcal{S}_s F_2$.

The monitoring operators \uparrow , \downarrow , $[-, -)_s$, and $[-, -)_w$ were inspired by work in runtime verification in [127]. We found these operators often more intuitive and compact than the usual past time operators in specifying runtime requirements, despite the fact that they have the same expressive power as the standard ones, as we discovered later. The operator $\uparrow F$ should be read “start F ”; it says that the formula F just started to be true, that is, it was false previously but it is true now. Dually, the operator $\downarrow F$ which is read “end F ”, carries the intuition that F ends to be true, that is, it was previously true but it is false now. The operators $[F_1, F_2)_s$ and $[F_1, F_2)_w$ are read “strong/weak interval F_1, F_2 ” and they carry the intuition that F_1 was true at some point in the past but F_2 has not been seen to be true since then, including that moment. For example, if **START** and **DOWN** are predicates on the state of a web server to be monitored, then $[\text{START}, \text{DOWN})_s$ is a property stating that the server *was* rebooted recently and since then it was not down,

$t \models \text{true}$	is always true,
$t \models \text{false}$	is always false,
$t \models a$	iff $a(s_n)$ holds,
$t \models \neg F$	iff $t \not\models F$,
$t \models F_1 \text{ op } F_2$	iff $t \models F_1$ and/or/etc. $t \models F_2$, when op is \wedge/\vee /etc.,
$t \models \odot F$	iff $t' \models F$, where $t' = t_{n-1}$ if $n > 1$ and $t' = t$ if $n = 1$,
$t \models \diamond F$	iff $t_i \models F$ for some $1 \leq i \leq n$,
$t \models \Box F$	iff $t_i \models F$ for all $1 \leq i \leq n$,
$t \models F_1 \mathcal{S}_s F_2$	iff $t_j \models F_2$ for some $1 \leq j \leq n$ and $t_i \models F_1$ for all $j < i \leq n$,
$t \models F_1 \mathcal{S}_w F_2$	iff $t \models F_1 \mathcal{S}_s F_2$ or $t \models \Box F_1$,
$t \models \uparrow F$	iff $t \models F$ and $t_{n-1} \not\models F$,
$t \models \downarrow F$	iff $t_{n-1} \models F$ and $t \not\models F$,
$t \models [F_1, F_2]_s$	iff $t_j \models F_1$ for some $1 \leq j \leq n$ and $t_i \not\models F_2$ for all $j \leq i \leq n$,
$t \models [F_1, F_2]_w$	iff $t \models [F_1, F_2]_s$ or $t \models \Box \neg F_2$.

Figure 10.3: Semantics of finite trace past time LTL.

while $[\text{START}, \text{DOWN}]_w$ says that the server was not down recently, meaning that it was either not down at all recently or it was rebooted and since then it was not down.

10.3.2 Formal Semantics

We next present formally the intuitive semantics described above. We regard a trace as a finite sequence of abstract states. In practice, these states are generated by events emitted by the program or system that we want to observe. Such events could indicate when variables' values are changed or when locks are acquired or released by threads or processes, or even when a physical action takes place, such as opening or closing a valve, a gate, or a door. If s is a state and a is an atomic proposition then $a(s)$ is true if and only if a holds in the state s . Notice that we are loose with respect to what “holds” means, because, depending on the context, it can mean anything. However, in the case of JPAX the atomic predicates are just any Java boolean expressions and their satisfaction is decided by evaluating them in the current state of the Java program. If $t = s_1 s_2 \dots s_n$ ($n \geq 1$) is a trace then we let t_i denote the trace $s_1 s_2 \dots s_i$ for each $1 \leq i \leq n$. The formal semantics of the operators defined in the previous subsection is given in Figure 10.3.

$$\begin{array}{ll}
t \models \diamond F & \text{iff } t \models F \text{ or } (n > 1 \text{ and } t_{n-1} \models \diamond F), \\
t \models \Box F & \text{iff } t \models F \text{ and } (n > 1 \text{ implies } t_{n-1} \models \Box F), \\
t \models F_1 \mathcal{S}_s F_2 & \text{iff } t \models F_2 \text{ or } (n > 1 \text{ and } t \models F_1 \text{ and } t_{n-1} \models F_1 \mathcal{S}_s F_2), \\
t \models F_1 \mathcal{S}_w F_2 & \text{iff } t \models F_2 \text{ or } (t \models F_1 \text{ and } (n > 1 \text{ implies } t_{n-1} \models F_1 \mathcal{S}_w F_2)), \\
t \models [F_1, F_2]_s & \text{iff } t \not\models F_2 \text{ and } (t \models F_1 \text{ or } (n > 1 \text{ and } t_{n-1} \models [F_1, F_2]_s)), \\
t \models [F_1, F_2]_w & \text{iff } t \not\models F_2 \text{ and } (t \models F_1 \text{ or } (n > 1 \text{ implies } t_{n-1} \models [F_1, F_2]_w)).
\end{array}$$

Figure 10.4: Recursive semantics of finite trace past time LTL.

Notice the special semantics of the operator “previously ” on a trace of one state: $s \models \circ F$ iff $s \models F$. This is consistent with the view that a trace consisting of exactly one state s is considered like a *stationary* infinite trace containing only the state s . We adopted this view because of intuitions related to monitoring. One can start monitoring a process potentially at any moment, so the first state in the trace might be different from the initial state of the monitored process. We think that the “best guess” one can have w.r.t. the past of the monitored program is that it was stationary. Alternatively, one could consider that $\circ F$ is false on a trace of one state for any atomic proposition F , but we find this semantics inconvenient because some atomic propositions may be related, such as, for example, a proposition “gate-up” and a proposition “gate-down”.

10.3.3 Recursive Semantics

An observation of crucial importance in the design of the subsequent algorithms is that the semantics above can be defined recursively, in such a way that the satisfaction relation for a formula and a trace can be calculated along the execution trace looking only one step backwards, as shown in Figure 10.4.

For example, according to the formal, nonrecursive, semantics, a trace $t = s_1 s_2 \dots s_n$ satisfies the formula $[F_1, F_2]_w$ if and only if either F_2 was false all the time in the past or otherwise F_1 was true at some point and since then F_2 was always false, including that moment. Therefore, in the case of a trace of size 1, i.e., when $n = 1$, it follows immediately that $t \models [F_1, F_2]_w$ if and only if $t \not\models F_2$. Otherwise, if the trace has more than one event then first of all $t \not\models F_2$, and then either $t \models F_1$ or else the prefix trace satisfies the interval formula, that is, $t_{n-1} \models [F_1, F_2]_w$. Similar reasoning applies to the other recurrences.

10.3.4 Equivalent Logics

We call the past time temporal logic presented above $ptLTL$. There is a tendency among logicians to minimize the number of operators in a given logic. For example, it is known that two operators are sufficient in propositional calculus, and two more (“next” and “until”) are needed for future time temporal logics. There are also various ways to minimize $ptLTL$. Let $ptLTL|_{Ops}$ be the restriction of $ptLTL$ to the propositional operators plus the operations in Ops . Then

Theorem 19 *The following 12 logics are all equivalent to $ptLTL$:*

1. $ptLTL|_{\{\ominus, \mathcal{S}_s\}}$,
2. $ptLTL|_{\{\ominus, \mathcal{S}_w\}}$,
3. $ptLTL|_{\{\ominus, []_s\}}$,
4. $ptLTL|_{\{\ominus, []_w\}}$,
5. $ptLTL|_{\{\uparrow, \mathcal{S}_s\}}$,
6. $ptLTL|_{\{\uparrow, \mathcal{S}_w\}}$,
7. $ptLTL|_{\{\uparrow, []_s\}}$,
8. $ptLTL|_{\{\uparrow, []_w\}}$,
9. $ptLTL|_{\{\downarrow, \mathcal{S}_s\}}$,
10. $ptLTL|_{\{\downarrow, \mathcal{S}_w\}}$,
11. $ptLTL|_{\{\downarrow, []_s\}}$,
12. $ptLTL|_{\{\downarrow, []_w\}}$.

The first two are known in the literature [128].

Proof: We first show the following properties:

1.	$\diamond F$	$=$	$true \mathcal{S}_s F$
2.	$\Box F$	$=$	$\neg \diamond \neg F$
3.	$F_1 \mathcal{S}_w F_2$	$=$	$(\Box F_1) \vee (F_1 \mathcal{S}_s F_2)$
4.	$\Box F$	$=$	$F \mathcal{S}_w false$
5.	$\diamond F$	$=$	$\neg \Box \neg F$
6.	$F_1 \mathcal{S}_s F_2$	$=$	$(\diamond F_2) \wedge (F_1 \mathcal{S}_w F_2)$
7.	$\uparrow F$	$=$	$F \wedge \neg \circ F$
8.	$\downarrow F$	$=$	$\neg F \wedge \circ F$
9.	$[F_1, F_2]_s$	$=$	$\neg F_2 \wedge ((\circ \neg F_2) \mathcal{S}_s F_1)$
10.	$[F_1, F_2]_w$	$=$	$\neg F_2 \wedge ((\circ \neg F_2) \mathcal{S}_w F_1)$
11.	$\downarrow F$	$=$	$\uparrow \neg F$
12.	$\uparrow F$	$=$	$\downarrow \neg F$
13.	$[F_1, F_2]_w$	$=$	$(\Box \neg F_2) \vee [F_1, F_2]_s$
14.	$[F_1, F_2]_s$	$=$	$(\diamond F_1) \wedge [F_1, F_2]_w$
15.	$\circ F$	$=$	$(F \rightarrow \neg \uparrow F) \wedge (\neg F \rightarrow \downarrow F)$
16.	$F_1 \mathcal{S}_s F_2$	$=$	$F_2 \vee [\circ F_2, \neg F_1]_s$

These properties are intuitive and relatively easy to prove. For example, property 15., the definition of $\circ F$ in terms of $\uparrow F$ and $\downarrow F$, says that in order to find out the value of a formula F in the previous state it suffices to look at the value of the formula in the current state and then, if it is true then look if the formula just started to be true or else look if the formula just ended to be true. We next only prove property 10., the proofs of the others are similar and straightforward.

In order to prove 10., one needs to show that for any trace t , it is the case that $t \models [F_1, F_2]_w$ if and only if $t \models_w \neg F_2 \wedge ((\circ \neg F_2) \mathcal{S}_w F_1)$. We show this by induction on the size of the trace t . If the size of t is 1, that is, if $t = s_1$, then

$$\begin{aligned}
t \models [F_1, F_2]_w &\text{ iff} \\
&\text{iff } t \not\models F_2 \\
&\text{iff } t \models \neg F_2 \\
&\text{iff (by "absorption" in boolean reasoning)} \\
&\quad t \models \neg F_2 \text{ and } (t \models F_1 \text{ or } t \models \neg F_2) \\
&\text{iff } t \models \neg F_2 \text{ and } (t \models F_1 \text{ or } t \models \circ \neg F_2) \\
&\text{iff } t \models \neg F_2 \wedge ((\circ \neg F_2) \mathcal{S}_w F_1).
\end{aligned}$$

If the size of the trace t is $n > 1$ then

$$\begin{aligned}
t \models [F_1, F_2]_w & \text{ iff} \\
& \text{iff (by the recursive semantics)} \\
& t \not\models F_2 \text{ and } (t \models F_1 \text{ or } \\
& \quad t_{n-1} \models [F_1, F_2]_w) \\
& \text{iff (by the induction hypothesis)} \\
& t \not\models F_2 \text{ and } (t \models F_1 \text{ or } \\
& \quad t_{n-1} \models \neg F_2 \wedge ((\odot \neg F_2) \mathcal{S}_w F_1)) \\
& \text{iff } t \not\models F_2 \text{ and } (t \models F_1 \text{ or } t_{n-1} \models \neg F_2 \text{ and } \\
& \quad t_{n-1} \models (\odot \neg F_2) \mathcal{S}_w F_1) \\
& \text{iff } t \not\models F_2 \text{ and } (t \models F_1 \text{ or } t \models \odot \neg F_2 \text{ and } \\
& \quad t_{n-1} \models (\odot \neg F_2) \mathcal{S}_w F_1) \\
& \text{iff (by the recursive semantics)} \\
& t \not\models F_2 \text{ and } t \models (\odot \neg F_2) \mathcal{S}_w F_1 \\
& \text{iff } t \models \neg F_2 \wedge ((\odot \neg F_2) \mathcal{S}_w F_1).
\end{aligned}$$

Therefore, $[F_1, F_2]_w = \neg F_2 \wedge ((\odot \neg F_2) \mathcal{S}_w F_1)$.

The equivalences of the 12 logics with *ptLTL* follow now immediately. For example, in order to show the 8th logic, $ptLTL|_{\{\uparrow, \downarrow, \odot\}}$, equivalent to *ptLTL*, one needs to show how the operators \uparrow and $[-, _]_s$ can define all the other past time temporal operators. This is straightforward because, 11. shows how \downarrow can be defined in terms of \uparrow , 15. shows how $\odot F$ can be defined using just \uparrow and \downarrow , 16. defines \mathcal{S}_s , 1. defines \diamond , 2. \Box , 3. \mathcal{S}_w , and 13. defines the weak interval. The interested reader can check the other 11 equivalences of logics. \square

Unlike in theoretical research, in practical monitoring of programs we want to have as many temporal operators available as possible and *not* to automatically translate them into a reduced kernel set. The reason is twofold. On the one hand, the more operators are available, the more succinct and natural the task of writing requirement specifications. On the other hand, as seen later in the paper, additional memory is needed for each temporal operator, so we want to keep the formulae as concise as possible.

10.4 Monitoring Safety by Rewriting

The architecture of JPAX is such that events extracted from a running program are sent to an observer which decides whether requirements are violated or not. An important concern that we had and are still having at this

relatively incipient stage of JPAX, is whether the chosen monitoring logics are expressive enough to specify powerful, practical and interesting requirements. Since flexibility with respect to defining/modifying monitoring logics is a very important factor at this stage, we have developed a rewriting-based framework which allows one to easily and effectively define new logics for runtime analysis and to monitor execution traces against formulae in these logics. We use the rewriting system Maude as a basis for this framework. In the following we first present Maude, and how formulae and important data structures are represented in Maude. Then we describe how basic propositional calculus is defined in Maude, and then how *ptLTL* is defined. Finally, it is described how this Maude definition of *ptLTL* is used for monitoring requirements stated by the user on execution traces.

10.4.1 Maude

We have implemented our logic-defining framework by rewriting in Maude [45, 47, 48]. Maude is a modularized membership equational [137] and rewriting logic [136] specification and verification system, whose operational engine is mainly based on a very efficient implementation of rewriting. A Maude module consists of sort and operator declarations, as well as equations relating terms over the operators and universally quantified variables. Modules can be composed in a hierarchical manner, building new theories from old theories. A particular attractive aspect of Maude is its *mix-fix* notation for syntax, which, together with precedence attributes of operators gives us an elegant way to compactly define syntax of logics. For example, the operation declarations²:

```

op _/\_ : Expression Expression
      -> Expression [prec 33] .
op _\/_ : Expression Expression
      -> Expression [prec 40] .
op [_,_] : Expression Expression
      -> Expression .
op if_then_else_ : Bool Expression Expression
      -> Expression .

```

define a simple syntax over the sort `Expression`, where conjunction and disjunction are infix operators (the underscores stand for arguments, whose

²These declarations are artificial and intended to explain some of Maude's features; they will not be needed later in the paper.

sorts are listed after the colon), while the interval and the conditional are mix-fix: operator and arguments can be mixed. Conjunction binds tighter than disjunction because it has a lower precedence (the lower the precedence the tighter the binding), so one is relieved from having to add useless parentheses to one's formulae.

It is often the case that equational and/or rewriting logics act like foundational logics, in the sense that other logics, or more precisely their syntax and operational semantics, can be expressed and efficiently executed by rewriting, so we regard Maude as a good choice to develop and prototype with various monitoring logics. The Maude implementations of the current logics supported by JPAX are quite compact. They are based on a simple, general architecture to define new logics which we only describe informally in the next subsection. Maude's notation will be introduced "on the fly" as needed.

10.4.2 Formulae and Data Structures

We have defined a generic module, called `FORMULA`, which defines the infrastructure for all the user-defined logics. Its Maude code is rather technical and so will not be given here. The module `FORMULA` includes some designated basic sorts, such as `Formula` for syntactic formulae, `FormulaDS` for formula data structures needed when more information than the formula itself should be stored for the next transition as in the case of past time LTL, `Atom` for atomic propositions (or state variables), `AtomState` for assignments of boolean values to atoms, also called "states", and `AtomState*` for such assignments together with *final* assignments, i.e., those that are followed by the end of a trace, sometimes requiring a special evaluation procedure. A state `As` is made terminal by applying it a unary operator, `_* : AtomState -> AtomState*`. `Formula` is a subsort of `FormulaDS`, because there are logics in which no extra information but a modified formula needs to be carried over for the next iteration (such as future time LTL which is also provided by JPAX). There are two constants of sort `Formula` provided, namely `true` and `false`, with the obvious meaning. The propositions that hold in a certain program state are generated by the executing instrumented program.

One of the most important operators in `FORMULA` is `_{}:FormulaDS AtomState* -> FormulaDS`, which updates the formula data structure when an (abstract) state change occurs during the execution of the program. Notice the use of mix-fix notation for operator declaration, in which underscores represent places of arguments, their order being the one in the arity of the operator.

On atomic propositions, say A , the module `FORMULA` defines the “update” operator as follows: $A\{As^*\}$ is true or false, depending on whether As^* assigns true or false to the atom A , where As^* is an atom state (i.e., an assignment from atoms to boolean values), which is either a terminal state (the last in a trace) or not. In the case of propositional calculus, this update operation basically evaluates propositions in the new state. For other logics it can be more complicated, depending on their trace semantics.

10.4.3 Propositional Calculus

Propositional calculus should be included in any monitoring logic. Therefore, we begin with the following module which is heavily used in JPAX. It implements an efficient rewriting procedure due to Hsiang [105] to decide validity of propositions, reducing any boolean expression to an exclusive disjunction (formally written $_++_$) of conjunctions ($_/__$):

```
fmod PROP-CALC is extending FORMULA .
*** Constructors ***
  op _/\_ : Formula Formula
    -> Formula [assoc comm] .
  op _++_ : Formula Formula
    -> Formula [assoc comm] .

  vars X Y Z : Formula . var As* : AtomState* .
  eq true /\ X = X .
  eq false /\ X = false .
  eq false ++ X = X .
  eq X ++ X = false .
  eq X /\ X = X .
  eq X /\ (Y ++ Z) = (X /\ Y) ++ (X /\ Z) .

*** Derived operators ***
  op _\/_ : Formula Formula -> Formula .
  op _->_ : Formula Formula -> Formula .
  op _<->_ : Formula Formula -> Formula .
  op !_ : Formula -> Formula .
  eq X \/_ Y = (X /\ Y) ++ X ++ Y .
  eq ! X = true ++ X .
  eq X -> Y = true ++ X ++ (X /\ Y) .
  eq X <-> Y = true ++ X ++ Y .

*** Operational Semantics
```

```

eq (X /\ Y){As*} = X{As*} /\ Y{As*} .
eq (X ++ Y){As*} = X{As*} ++ Y{As*}
endfm

```

In Maude, operators are introduced after the **op** and **ops** (when more than one operator is introduced) symbols. Operators can be given attributes in square brackets, such as associativity and commutativity. Universally quantified variables used in equations are introduced after the **var** and **vars** symbols. Finally, equations are introduced after the **eq** symbol. The specification of the simple propositional calculus above shows the flexibility of the mix-fix notation of Maude, which allows us to define the syntax of a logic in the most natural way.

The equations above are interpreted as rewriting rules by Maude, so they will be applied from left to right only. However, due to the associativity and commutativity attributes, rewrites as well as matchings are applied *modulo* associativity and commutativity (AC), making therefore the procedure implied by the rewrite rules for propositional calculus above highly non-trivial. As proved by Hsiang [105], the AC rewriting system above has the property that any proposition is reduced to true or false if it is semantically true or false, or otherwise to a canonical form modulo AC; thus two formulae are equivalent if and only if their canonical forms are equal modulo AC. We found this procedure quite convenient so far, being able to efficiently reduce formulae of hundreds of symbols that occurred in practical examples. However, one should of course not expect this procedure to work efficiently on any proposition, because the propositional validity problem is NP-complete.

10.4.4 Past Time Linear Temporal Logic

Past time LTL can now be implemented on top of the provided logic-defining framework. Our rewriting based implementation below follows the recursive semantics of past time LTL defined in Subsection 10.3.3, and, it appears similar to the Java implementation used in [127]. We next explain the PT-LTL module in detail.

We start by defining the syntax of past time LTL. Since it extends the module **PROP-CALC** of propositional calculus, we only have to define syntax for the temporal operators:

```

fmod PT-LTL is extending PROP-CALC .
  op (*)_ : Formula -> Formula .

```

```

*** previously
op <*_ : Formula -> Formula .
*** eventually in the past
op [*]_ : Formula -> Formula .
*** always in the past
op _Ss_ : Formula Formula -> Formula .
*** strong since
op _Sw_ : Formula Formula -> Formula .
*** weak since
op start : Formula -> Formula .
*** start
op end : Formula -> Formula .
*** end
op [_,_]s : Formula Formula -> Formula .
*** strong interval
op [_,_]w : Formula Formula -> Formula .
*** weak interval

```

We have used a curly bracket to close the intervals because, for some technical parsing related reasons, Maude does not allow unbalanced parentheses in its terms. The syntax above can now be used by users to write monitoring requirements as formulae. These formulae are loaded by JPAX at initialization and then sent to Maude for parsing and processing. When the first event from the instrumented program is received by JPAX, it sends this event to Maude in order to initialize its monitoring data structures associated to its formulae (remember that the recursive definition of past time LTL in Subsection 10.3.3 treats the first event of the trace differently). This is done by launching the reduction `mkDS(F, As)` in Maude, where `F` is the formula to monitor and `As` is the atom state abstracting the first event generated by the monitored program; `mkDS` is an abbreviation for “make data structure” and is defined below.

Before we define the operation `mkDS`, we first discuss the formula data structures storing not only the formulae but also their current satisfaction status. It is worth noticing that the strong and weak temporal operators have exactly the same recursive semantics starting with the second event. That suggests that we do not need nodes of different type (strong and weak) in the formula data structure once the monitoring process is initialized: the difference between strong and weak versions of an operator are rather represented by the initial values passed as arguments to a single common version of the operator. The following operation declarations therefore define the constructors for these data structures:

```

op atom          : Atom Bool -> FormulaDS .
op and   : FormulaDS FormulaDS Bool -> FormulaDS .
op xor   : FormulaDS FormulaDS Bool -> FormulaDS .
op previously : FormulaDS Bool -> FormulaDS .
op eventuallyPast : FormulaDS Bool -> FormulaDS .
op alwaysPast   : FormulaDS Bool -> FormulaDS .
op since  : FormulaDS FormulaDS Bool -> FormulaDS .
op start   : FormulaDS Bool -> FormulaDS .
op end     : FormulaDS Bool -> FormulaDS .
op interval: FormulaDS FormulaDS Bool -> FormulaDS .

```

The first operation defines a cell storing an atomic proposition together with its observed boolean value, while the next two store conjunction and exclusive disjunction nodes. According to the propositional calculus procedure defined in module PROP-CALC in Subsection 10.4.3, these are the only propositional operators that can occur in reduced formulae. The remaining operators are the seven past time temporal operators introduced so far.

An operator that extracts the boolean value associated to a temporal formula is needed in the sequel, so we define it next. The syntax of this operator is `[_] : FormulaDS -> Bool` and it is defined in the module FORMULA, together with its obvious equations `[true] = true` and `[false] = false`. Its definition on temporal and propositional and temporal operators follows:

```

var A : Atom . var B : Bool .
vars D Dx Dy : FormulaDS .
eq [and(Dx,Dy,B)] = B .
eq [xor(Dx,Dy,B)] = B .
eq [atom(A,B)] = B .
eq [previously(D,B)] = B .
eq [eventuallyPast(D,B)] = B .
eq [alwaysPast(D,B)] = B .
eq [since(Dx,Dy,B)] = B .
eq [interval(Dx,Dy,B)] = B .
eq [start(Dx,B)] = B .
eq [end(Dx,B)] = B .

```

The operation `mkDS` can be defined now. It basically follows the recursive semantics in Subsection 10.3.3, when the length of the trace is 1:

```

vars X Y : Formula .
op mkDS : Formula AtomState -> FormulaDS .
eq mkDS(true, As) = true .

```

```

eq mkDS(false, As) = false .
eq mkDS(A, As) = atom(A, (A{As} == true)) .
eq mkDS(X /\ Y, As) =
  and(mkDS(X,As), mkDS(Y,As),
    [mkDS(X,As)] and [mkDS(Y,As)]) .
eq mkDS(X ++ Y, As) =
  xor(mkDS(X,As), mkDS(Y,As),
    [mkDS(X,As)] xor [mkDS(Y,As)]) .
eq mkDS( (*)X, As) =
  previously(mkDS(X, As), [mkDS(X, As)]) .
eq mkDS( <*>X, As) =
  eventuallyPast(mkDS(X, As), [mkDS(X, As)]) .
eq mkDS( [*]X, As) =
  alwaysPast(mkDS(X, As), [mkDS(X, As)]) .
eq mkDS(X Ss Y, As) =
  since(mkDS(X,As), mkDS(Y,As), [mkDS(Y,As)]) .
eq mkDS(X Sw Y, As) =
  since(mkDS(X,As), mkDS(Y,As),
    [mkDS(X,As)] or [mkDS(Y,As)]) .
eq mkDS(start(X), As) = start(mkDS(X,As),false) .
eq mkDS(end(X), As) = end(mkDS(X,As), false) .
eq mkDS([X,Y]s, As) =
  interval(mkDS(X,As), mkDS(Y,As),
    [mkDS(Y,As)] and not [mkDS(X,As)]) .
eq mkDS([X,Y]w, As) =
  interval(mkDS(X,As), mkDS(Y,As),
    not [mkDS(Y,As)]) .

```

The data structure associated to a past time formula is essentially its syntax tree augmented with a boolean bit for each node. Each boolean bit will store the result of the satisfaction relation between the current execution trace and the corresponding subformula. The only thing left is to define how the formula data structures, or more precisely their bits, modify when a new event is received. This is defined below, using the operator $_ \{ _ \}$: $\text{FormulaDS AtomState} \rightarrow \text{FormulaDS}$ provided by the module `Formula`:

```

eq atom(A, B){As} = atom(A, (A{As} == true)) .
eq and(Dx, Dy, B){As} =
  and(Dx{As}, Dy{As}, [Dx{As}] and [Dy{As}]) .
eq xor(Dx, Dy, B){As} =
  xor(Dx{As}, Dy{As}, [Dx{As}] xor [Dy{As}]) .
eq previously(D,B){As} = previously(D{As}, [D]) .
eq eventuallyPast(D, B){As} =

```



```

      eventuallyPast(D{As}, [D{As}] or B) .
eq alwaysPast(D, B){As} =
  alwaysPast(D{As}, [D{As}] and B) .
eq since(Dx, Dy, B){As} =
  since(Dx{As}, Dy{As},
    [Dy{As}] or [Dx{As}] and B) .
eq start(Dx,B){As} =
  start(Dx{As}, [Dx{As}] and not B) .
eq end(Dx,B){As} =
  end(Dx{As}, not [Dx{As}] and B) .
eq interval(Dx, Dy, B){As} =
  interval(Dx{As}, Dy{As}, not [Dy{As}] and
    ([Dx{As}] or B)) .
endfm

```

The operator `_==_` is built-in and takes two terms of same sort, reduces them to their normal forms, and then returns true if they are equal and false otherwise.

10.4.5 Monitoring with Maude

In this subsection we give more details on how the actual rewriting based monitoring process works. When the JPAX system is started, the user is supposed to have already specified several formulae in a file containing monitoring requirements. The first thing JPAX does is to start a Maude process, load the past time LTL semantics described above, and then set Maude to run in its *loop mode*, which is an execution mode in which Maude maintains a state term which the user (potentially another process, such as JPAX) can modify interactively. Then JPAX sends Maude all the requirement formulae that the user wants to monitor. Maude stores them in its loop state and waits for JPAX to send events. Notice that the above is general and applies to any logic.

When JPAX receives the first event from the instrumented program that is relevant for the past time LTL analysis module, it just sends it to Maude. On receiving the first event, say `As`, Maude needs to generate the formula data structures for all the formulae to be monitored. It does so by replacing each formula `F` in the loop state by the normal form of the term `mkDS(F, As)`. Then it waits for JPAX to submit further events. Each time a new relevant event `As` is received by JPAX from the instrumented program, it just forwards it to Maude. Then Maude replaces each formula data structure `D` in its loop state by `D{As}` and then waits for further events.

If at any moment $[D]$ is false for the data structure D associated to a formula F , then Maude sends an error message to JPAX, which further warns the user appropriately.

It should be obvious that the runtime complexity of the rewriting monitoring algorithm is $O(m)$ to process an event, where m is the size of the past time LTL formula to monitor. That is, the algorithm only needs to traverse the data structure representing the formula bottom-up for each new event, and update one bit in each node. So the overall runtime complexity is $O(n \cdot m)$, where n is the number of events to be monitored. This is the best one can asymptotically hope from a runtime monitoring algorithm, but of course, there is room for even faster algorithms in practical situations, as the one presented in the next section. The main benefit of the rewriting algorithm presented in this section is that it falls under the general framework by which one can easily add or experiment with new monitoring logics within the JPAX system.

The Maude code performing the above steps is relatively straightforward but rather ugly, so we prefer not to present it here. Additionally, Maude's support for inter-process communication is planned to be changed soon, so this code would become soon obsolete.

10.5 Synthesizing Monitors for Safety Properties

The rewriting algorithm above is a very good choice in the context of the current version of JPAX, because it gives us flexibility and is efficient enough to process events at a faster rate than they can actually be sent by JPAX. However, there might be situations in which a full scale AC rewriting engine like Maude is not available, such as within an embedded system, or in which as little runtime overhead as possible is allowed, such as in real time applications. In this section we present a dynamic programming based algorithm, also based on the recursive semantics of past time LTL in Subsection 10.3.3, which takes as input a formula and generates source code which can further be compiled into an efficient executable monitor for that formula. This algorithm can be used in two different ways. On the one hand, it can be used as an efficient external monitor to take an action when a formula is violated, such as to report an error to a user, to reboot the system, to send a message, or even to generate a correcting task. On the other hand, it can be used in a context in which one allows past time LTL annotations in the source code of a program, where the logical annotations

can be expanded into source code which is further compiled together with the original program. These two use modes, offline versus inline, are further explained in Subsection 10.5.4.

10.5.1 The Algorithm Illustrated by an Example

In this section we show via an example how to generate dynamic programming code for a concrete *ptLTL*-formula. We think that this example would practically be sufficient for the reader to foresee our general algorithm presented in the next subsection. Let $\uparrow p \rightarrow [q, \downarrow (r \vee s)]_s$ be the *ptLTL*-formula that we want to generate code for. The formula states: “whenever p becomes true, then q has been true in the past, and since then we have not yet seen the end of r or s ”. The code translation depends on an enumeration of the subformulae of the formula that satisfies the *enumeration invariant*: any formula has an enumeration number smaller than the numbers of all its subformulae. Let $\varphi_0, \varphi_1, \dots, \varphi_8$ be such an enumeration:

$$\begin{aligned}\varphi_0 &= \uparrow p \rightarrow [q, \downarrow (r \vee s)]_s, \\ \varphi_1 &= \uparrow p, \\ \varphi_2 &= p, \\ \varphi_3 &= [q, \downarrow (r \vee s)]_s, \\ \varphi_4 &= q, \\ \varphi_5 &= \downarrow (r \vee s), \\ \varphi_6 &= r \vee s, \\ \varphi_7 &= r, \\ \varphi_8 &= s.\end{aligned}$$

Note that the formulae have here been enumerated in a post-order fashion. One could have chosen a breadth-first order, or any other enumeration, as long as the enumeration invariant is true.

The input to the generated program will be a finite trace $t = s_1 s_2 \dots s_n$ of n events. The generated program will maintain a state via a function $update : \mathbf{State} \times Event \rightarrow \mathbf{State}$, which updates the state with a given event.

In order to illustrate the dynamic programming aspect of the solution, one can imagine recursively defining a matrix $s[1..n, 0..8]$ of boolean values $\{0, 1\}$, with the meaning that $s[i, j] = 1$ iff $t_i \models \varphi_j$. Then one can fill the table according to the recursive semantics of past time LTL as described in Subsection 10.3.3. This would be the standard way of regarding the above satisfaction problem as a dynamic programming problem. An important

observation is, however, that, like in many other dynamic programming algorithms, one doesn't have to store the entire table $s[1..n, 0..8]$, which would be quite large in practice; in this case, one needs only $s[i, 0..8]$ and $s[i-1, 0..8]$, which we'll write $now[0..8]$ and $pre[0..8]$ from now on, respectively. It is now only a relatively simple exercise to write up the following algorithm for checking the above formula on a finite trace:

```

State  $state \leftarrow \{\}$ ;

bit  $pre[0..8]$ ;

bit  $now[0..8]$ ;

INPUT: trace  $t = s_1 s_2 \dots s_n$ ;

/* Initialization of  $state$  and  $pre$  */

 $state \leftarrow update(state, s_1)$ ;

 $pre[8] \leftarrow s(state)$ ;

 $pre[7] \leftarrow r(state)$ ;

 $pre[6] \leftarrow pre[7] \text{ or } pre[8]$ ;

 $pre[5] \leftarrow \text{false}$ ;

 $pre[4] \leftarrow q(state)$ ;

 $pre[3] \leftarrow pre[4] \text{ and not } pre[5]$ ;

 $pre[2] \leftarrow p(state)$ ;

 $pre[1] \leftarrow \text{false}$ ;

 $pre[0] \leftarrow \text{not } pre[1] \text{ or } pre[3]$ ;

/* Event interpretation loop */

for  $i = 2$  to  $n$  do {

     $state \leftarrow update(state, s_i)$ ;

     $now[8] \leftarrow s(state)$ ;

     $now[7] \leftarrow r(state)$ ;

```

```

    now[6] ← now[7] or now[8];
    now[5] ← not now[6] and pre[6];
    now[4] ← q(state);
    now[3] ← (pre[3] or now[4]) and not now[5];
    now[2] ← p(state);
    now[1] ← now[2] and not pre[2];
    now[0] ← not now[1] or now[3];

    if now[0] = 0 then
        output('‘property violated’');
        pre ← now;
    };

```

In the following we explain the generated program.

Declarations Initially a state is declared. This will be updated as the input event list is processed. Next, the two arrays *pre* and *now* are declared. The *pre* array will contain values of all subformulae in the previous state, while *now* will contain the value of all subformulae in the current state.

Initialization The initialization phase consists of initializing the *state* variable and the *pre* array. The first event s_1 of the event list is used to initialize the *state* variable. The *pre* array is initialized by evaluating all subformulae bottom up, starting with highest formula numbers, and assigning these values to the corresponding elements of the *pre* array; hence, for any $i \in \{0 \dots 8\}$ $pre[i]$ is assigned the initial value of formula φ_i . The *pre* array is initialized in such a way as to maintain the view that the initial state is supposed stationary before monitoring is started. This in particular means that $\uparrow p$ is false, as well as is $\downarrow (r \vee s)$, since there is no change in state (indices 1 and 5). The interval operator has the obvious initial interpretation: the first argument must be true and the second false for the formula to be true (index 3). Propositions are true if they hold in the initial state (indices 2, 4, 7 and 8), and boolean operators are interpreted the standard way (indices 0, 6).

Event Loop The main evaluation loop goes through the event trace, starting from the second event. For each such event, the state is updated, followed by assignments to the *now* array in a bottom-up fashion similar to the initialization of the *pre* array: the array elements are assigned values from higher index values to lower index values, corresponding to the values of the corresponding subformulae. Propositional boolean operators are interpreted the standard way (indices 0 and 6). The formula $\uparrow p$ is true if p is true now and not true in the previous state (index 1). Similarly with the formula $\downarrow (r \vee s)$ (index 5). The formula $[q, \downarrow (r \vee s)]_s$ is true if either the formula was true in the previous state, or q is true in the current state, and in addition $\downarrow (r \vee s)$ is not true in the current state (index 3). At the end of the loop an error message is issued if *now*[0], the value of the whole formula, has the value 0 in the current state. Finally, the entire *now* array is copied into *pre*.

Given a fixed *ptLTL* formula, the analysis of this algorithm is straightforward. Its time complexity is $\Theta(n)$ where n is the length of the input trace, the constant being given by the size of the *ptLTL* formula. The memory required is constant, since the length of the two arrays is the size of the *ptLTL* formula. However, one may want to also include the size of the formula, say m , into the analysis; then the time complexity is obviously $\Theta(n \cdot m)$ while memory required is $2 \cdot (m + 1)$ bits. The authors conjecture that it's hard to find an algorithm running faster than the above in practical situations, though some slight optimizations are possible (see Section 10.5.3).

10.5.2 The Algorithm Formalized

We now formally describe our algorithm that synthesizes a dynamic programming algorithm from a *ptLTL*-formula. It takes as input a formula and generates a program as the one above, containing a “for” loop which traverses the trace of events, while validating or invalidating the formula. The generated program is printed using the function **output**, which takes one or more string or integer parameters which are concatenated in the output. This algorithm is designed to generate pseudocode, but it can easily be adapted to generate code in any imperative programming language:

INPUT: past time LTL formula φ

let $\varphi_0, \varphi_1, \dots, \varphi_m$ be the subformulae of φ ;

```

output("State  $state \leftarrow \{\}$ ");
output("bit  $pre[0..m]$ ");
output("bit  $now[0..m]$ ");
output("INPUT: trace  $t = s_1s_2...s_n$ ");
output("/* Initialization of  $state$  and  $pre$  */");
output("state  $\leftarrow update(state, s_1)$ ");
for  $j = m$  downto 0 do {
    output("     $pre[$ ,  $j$ ,  $]$   $\leftarrow$  ");
    if  $\varphi_j$  is a variable then
        output( $\varphi_j$ , "(state)");
    if  $\varphi_j$  is true then output("true");
    if  $\varphi_j$  is false then output("false");
    if  $\varphi_j = \neg\varphi_{j'}$  then
        output("not  $pre[$ ,  $j'$ ,  $]$ ");
    if  $\varphi_j = \varphi_{j_1} \text{ op } \varphi_{j_2}$  then
        output("pre[ $$ ,  $j_1$ ,  $]$  op pre[ $$ ,  $j_2$ ,  $]$ ");
    if  $\varphi_j = \odot\varphi_{j_1}$  then
        output("pre[ $$ ,  $j_1$ ,  $]$ ");
    if  $\varphi_j = \diamond\varphi_{j_1}$  then
        output("pre[ $$ ,  $j_1$ ,  $]$ ");
    if  $\varphi_j = \Box\varphi_{j_1}$  then
        output("pre[ $$ ,  $j_1$ ,  $]$ ");
    if  $\varphi_j = \varphi_{j_1} S_s \varphi_{j_2}$  then
        output("pre[ $$ ,  $j_2$ ,  $]$ ");

```

```

if  $\varphi_j = \varphi_{j_1} S_w \varphi_{j_2}$  then
    output(“pre[”,  $j_1$ , “] or pre[”,  $j_2$ , “];”);
if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_s$  then
    output(“pre[”,  $j_1$ , “] and not pre[”,  $j_2$ , “];”);
if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_w$  then
    output(“not pre[”,  $j_2$ , “];”);
if  $\varphi_j = \uparrow \varphi_{j'}$  then output(“false;”);
if  $\varphi_j = \downarrow \varphi_{j'}$  then output(“false;”);
};
output(“/* Event interpretation loop */”);
output(“for  $i = 2$  to  $n$  do {”);
for  $j = m$  downto 0 do {
    output(“    now[”,  $j$ , “]  $\leftarrow$  ”);
    if  $\varphi_j$  is a variable then output( $\varphi_j$ , “(state);”);
    if  $\varphi_j$  is true then output(“true;”);
    if  $\varphi_j$  is false then output(“false;”);
    if  $\varphi_j = \neg \varphi_{j'}$  then output(“not now[”,  $j'$ , “];”);
    if  $\varphi_j = \varphi_{j_1} op \varphi_{j_2}$  then
        output(“now[”,  $j_1$ , “] op now[”,  $j_2$ , “];”);
    if  $\varphi_j = \odot \varphi_{j_1}$  then output(“pre[”,  $j_1$ , “];”);
    if  $\varphi_j = \diamond \varphi_{j_1}$  then
        output(“pre[”,  $j$ , “] or now[”,  $j_1$ , “]”);
    if  $\varphi_j = \Box \varphi_{j_1}$  then
        output(“pre[”,  $j$ , “] and now[”,  $j_1$ , “]”);

```



```

if  $\varphi_j = \varphi_{j_1} S_s \varphi_{j_2}$  then
  output(“(pre[”, j, “] and now[”, j1, “] or
    now[”, j2, “];”);
if  $\varphi_j = \varphi_{j_1} S_w \varphi_{j_2}$  then
  output(“(pre[”, j, “] and now[”, j1, “] or
    now[”, j2, “];”);
if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_s$  then
  output(“(pre[”, j, “] or now[”, j1, “] and
    not now[”, j2, “];”);
if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_w$  then
  output(“(pre[”, j, “] or now[”, j1, “] and
    not now[”, j2, “];”);
if  $\varphi_j = \uparrow \varphi_{j'}$  then
  output(“now[”, j', “] and
    not pre[”, j', “];”);
if  $\varphi_j = \downarrow \varphi_{j'}$  then
  output(“not now[”, j', “] and
    pre[”, j', “];”);
};
output(“    if now[0] = 0 then
  output(“‘property violated’”);
output(“    pre ← now;”);
output(“}”);

```

op is any binary propositional connective. Since we have already given a detailed explanation of the example in the previous section, we shall only give a very brief description of this algorithm.

The formula should be first visited top down to assign increasing numbers to subformulae as they are visited. Let $\varphi_0, \varphi_1, \dots, \varphi_m$ be the list of all

subformulae. Because of the recursive nature of *ptLTL*, this step ensures us that the truth value of $t_i \models \varphi_j$ can be completely determined from the truth values of $t_i \models \varphi_{j'}$ for all $j < j' \leq m$ and the truth values of $t_{i-1} \models \varphi_{j'}$ for all $j \leq j' \leq m$.

Before we generate the main loop, we should first generate code for initializing the array *pre*[0..*m*], basically giving it the truth values of the subformulae on the initial state, conceptually being an infinite trace with repeated occurrences of the initial state. After that, the generated main event loop will process the events. The loop body will update/calculate the array *now* and in the end will move it into the array *pre* to serve as basis for the next iteration. After each iteration *i*, *now*[0] tells whether the formula is validated by the trace $s_1s_2\dots s_i$.

Since the formula enumeration procedure is linear, the algorithm synthesizes a dynamic programming algorithm from an *ptLTL* formula in linear time with the size of the formula. The boolean operations used above are usually very efficiently implemented on any microprocessor and the arrays of bits *pre* and *now* are small enough to be kept in cache. Moreover, the dependencies between instructions in the generated “for” loop are simple to analyze, so a reasonable compiler can easily unfold or/and parallelize it to take advantage of machine’s resources. Consequently, the generated code is expected to run very fast. We shall next illustrate how such optimizations can be part of the translation algorithm.

10.5.3 Optimizing the Generated Code

The generated code presented in Subsection 10.5.1 is not optimal. Even though a smart compiler can in principle generate good machine code from it, it is still worth exploring ways to synthesize directly optimized code especially because there are some attributes that are specific to the runtime observer which a compiler cannot take into consideration.

A first observation is that not all the bits in *pre* are needed, but only those which are used at the next iteration, namely 2, 3, and 6. Therefore, only a bit per temporal operator is needed, thereby reducing significantly the memory required by the generated algorithm. Then the body of the generated “for” loop becomes after (blind) substitution (we don’t consider the initialization code here):

$$\begin{aligned} state &\leftarrow update(state, s_i) \\ now[3] &\leftarrow r(state) \text{ or } s(state) \end{aligned}$$

```

now[2] ← (pre[2] or q(state)) and
           not (not now[3] and pre[3])
now[1] ← p(state)
if ((not (now[1] and not pre[1]) or now[2]) = 0)
    then output(‘‘property violated’’);

```

that can be further optimized by boolean simplifications:

```

state ← update(state, si)
now[3] ← r(state) or s(state)
now[2] ← (pre[2] or q(state)) and
           (now[3] or not pre[3])
now[1] ← p(state)
if (now[1] and not pre[1] and not now[2])
    then output(‘‘property violated’’);

```

The most expensive part of the code above is the function calls, namely $p(state)$, $q(state)$, $r(state)$, and $s(state)$. Depending upon the runtime requirements, the execution time of these functions may vary significantly. However, since one of the major concerns of monitoring is to affect the normal execution of the monitored program as little as possible, especially in the inline monitoring approach, one would of course want to evaluate the atomic predicates on states only if really needed, or rather to evaluate only those that, probabilistically, add a minimum cost. Since we don't want to count on an optimizing compiler, we prefer to store the boolean formula as some kind of binary decision diagram, more precisely, as a term over the conditional operation $_{?} : _$, where ' $e_1 ? e_2 : e_3$ ' means: "*if e_1 then e_2 else e_3* ". For example, $pre[3] ? pre[2] ? now[3] : q(state) : pre[2] ? 1 : q(state)$ (see [84] for a formal definition). Therefore, one is faced with the following optimization problem:

Given a boolean formula φ using propositions a_1, a_2, \dots, a_n of costs c_1, c_2, \dots, c_n , respectively, find a $(_{?} : _)$ -expression that optimally implements φ .

We have implemented a procedure in Maude, on top of propositional calculus, which generates all correct $(_? _ : _)$ -expressions for φ , admittedly a potentially exponential number in the number of distinct atomic propositions in φ , and then chooses the shortest in size, ignoring the costs. Applied on the code above, it yields:

```

state ← update(state, si)

now[3] ← r(state) ? 1 : s(state)

now[2] ← pre[3] ? pre[2] ? now[3] :

      q(state) : pre[2] ? 1 : q(state)

now[1] ← p(state)

if (pre[1] ? 0 : now[2] ? 0 : now[1])

  then output(‘‘property violated’’);

```

We would like to extend our procedure to take the evaluation costs of predicates into consideration. These costs can either be provided by the user of the system or be calculated automatically by a static analysis of predicates’ code, or even be estimated by executing the predicates on a sample of states. However, based on our examples so far, we conjecture at this incipient stage that, given a boolean formula φ in which all the atomic propositions have the same cost, the probabilistically runtime optimal $(_? _ : _)$ -expression implementing φ is *exactly* the one which is smallest in size.

A further optimization would be to generate directly machine code instead of using a compiler. Then the arrays of bits *now* and *pre* can be stored in two registers, which would be all the memory needed. Since all the operations executed are bit operations, the generated code is expected to be very fast. One could even imagine hardware implementations of past time monitors, using the same ideas, in order to enforce safety requirements on physical devices.

10.5.4 Implementation of Offline and Inline Monitoring

In this section we briefly describe our efforts to implement the above described algorithm to create monitors for observing the execution of Java programs in PathExplorer. We present two approaches that we have pursued. In the *off-line* approach we create a monitor that runs in parallel with the

executing program, potentially on a different computer, receiving events from the running program, and checking on-the-fly that the formulae are satisfied. In this approach the formulae to be checked are given in a separate specification. In the *inline* approach, formulae are written as comments in the program text, and are then expanded into Java code that is inserted after the comments.

Offline Monitoring

The code generator for off-line monitoring has been written in Java, using JavaCC [110], an environment for writing parsers and for generating and manipulating abstract syntax trees. The input to the code generator is a specification given in a file separate from the program. The specification for our example looks as follows (the default interpretation of intervals is “strong”):

```
specification Example is
P = start(p) -> [q,end(r|s));
end
```

Several named formulae can be listed; here we have only included one, named P. The translator reads this specification and generates a single Java class, called **Formulae**, which contains all the machinery for evaluating all the formulae (in this case one) in the specification. This class must then be compiled and instantiated as part of the monitor. The class contains an **evaluate()** method which is applied after each state change and which will evaluate all the formulae. The class constructor takes as parameter a reference to the object that represents the state, such that any updates to the state by the monitor, based on received events, can be seen by the **evaluate()** method. The generated **Formulae** class for the above specification looks as follows:

```
class Formulae{
  abstract class Formula{
    protected String name;  protected State state;
    protected boolean[] pre; protected boolean[] now;

    public Formula(String name,State state){
      this.name = name; this.state = state;
    }
    public String getName(){return name;}
    public abstract boolean evaluate();
  }
  private List formulae = new ArrayList();
  public void evaluate(){
```

```

        Iterator it = formulae.iterator();
        while(it.hasNext()){
            Formula formula = (Formula)it.next();
            if(!formula.evaluate()){
                System.out.println("Property " + formula.getName() +
                                   " violated");
            }
        }
    }
    class Formula_P extends Formula{
        public boolean evaluate(){
            now[8] = state.holds("s");
            now[7] = state.holds("r");
            now[6] = now[7] || now[8];
            now[5] = !now[6] && pre[6];
            now[4] = state.holds("q");
            now[3] = (pre[3] || now[4]) && !now[5];
            now[2] = state.holds("p");
            now[1] = now[2] && !pre[2];
            now[0] = !now[1] || now[3];
            System.arraycopy(now,0,pre,0,9);
            return now[0];
        }
        public Formula_P(State state){
            super("P",state);
            pre = new boolean[9]; now = new boolean[9];
            pre[8] = state.holds("s");
            pre[7] = state.holds("r");
            pre[6] = pre[7] || pre[8];
            pre[5] = false;
            pre[4] = state.holds("q");
            pre[3] = pre[4] && !pre[5];
            pre[2] = state.holds("p");
            pre[1] = false;
            pre[0] = !pre[1] || pre[3];
        }
    }
    public Formulae(State state){
        formulae.add(new Formula_P(state));
    }
}

```

The class contains an inner abstract³ class **Formula** and, in the general case, an inner class **Formula_X** extending the **Formula** class for each formula in the specification, where **X** is the formula's name. In our case there is one such **Formula_P** class. The abstract **Formula** class declares the **pre** and **now** arrays, without giving them any size, since this is formula specific. An abstract **evaluate** method is also declared. The class **Formula_P** contains the real definition of this **evaluate()** method. The constructor for this class in addition initializes the sizes of **pre** and **now** depending on the size of the

³An abstract class is a class where some methods are abstract, by having no body. Implementations for these methods will be provided in extending subclasses.

formula, and also initializes the `pre` array.

In order to handle the general case where several formulae occur in the specification, and hence many `Formula_X` classes are defined, we need to create instances for all these classes and store them in some data structure where they can be accessed by the outermost `evaluate()` method. The `formulae` list variable is initialized to contain all these instances when the constructor of the `Formulae` class is called. The outermost `evaluate()` method, on each invocation, goes through this list and calls `evaluate()` on each single formula object.

Inline Monitoring

The general architecture of PAX was mainly designed for offline monitoring in order to accommodate applications where the source code is not available or where the monitored process is not even a program, but some kind of physical device. However, it is often the case that the source code of an application *is* available and that one is willing to accept extra code for testing purposes. Inline monitoring has actually higher precision because one knows exactly where an event was emitted in the execution of the program. Moreover, one can even throw exceptions when a safety property is violated, like in Temporal Rover [57], so the running program has the possibility to recover from an erroneous execution or to guide its execution in order to avoid undesired behaviors.

In order to provide support for inline monitoring, we developed some simple scripts that replace temporal annotations in Java source code by actual monitoring code, which throws an exception when the formula is violated. In [90] we show an example of expanded code for future time LTL. The “for” loop and the update of the state in the generic algorithm in Section 10.5.1 are not needed anymore because the atomic predicates use directly the current state of the program when the expanded code is reached during the execution. In [35] the tool Java-MoP is described, which implements the presented algorithm as a logic plug-in for inline monitoring (as well as for off-line monitoring).

The following code snippets illustrate the inline approach. Assume a class `A`, that defines four integer variables and a method `m`, which contains the past time temporal logic formula from above. Now the propositions `p`, `q`, `r` and `s` are defined to refer to the four variables. The intention is that whenever the program point of the comment is reached, the formula will be evaluated.

```

class A{
  int a,b,c,d;
  void m(){
    ...
    /* @monitor
       proposition p = a>0;
       proposition q = b>0;
       proposition r = c>0;
       proposition s = d>0;
       property P = start(p) -> [q,end(r|s));
    */
    ...
  }
}

```

This class is now automatically translated into the following, where code representing the semantics of the formula has been inserted at the position of the formula comment, and in the constructor:

```

class A{
  int a,b,c,d;
  boolean[] pre = new boolean[9];
  boolean[] now = new boolean[9];

  public A(){
    pre[8] = d>0;
    pre[7] = c>0;
    pre[6] = pre[7] || pre[8];
    pre[5] = false;
    pre[4] = b>0;
    pre[3] = pre[4] && !pre[5];
    pre[2] = a>0;
    pre[1] = false;
    pre[0] = !pre[1] || pre[3];
  }

  void m(){
    ...
    now[8] = d>0;
    now[7] = c>0;
    now[6] = now[7] || now[8];
    now[5] = !now[6] && pre[6];
    now[4] = b>0;
    now[3] = (pre[3] || now[4]) && !now[5];
    now[2] = a>0;
    now[1] = now[2] && !pre[2];
    now[0] = !now[1] || now[3];
    System.arraycopy(now,0,pre,0,9);
    if(!now[0])throw Violated("P");
    ...
  }
}

```


It is essentially the same code as in the offline case, except that the looping constructs have been removed.

It is inline monitoring that motivated us to optimize the generated code as much as possible as in Subsection 10.5.3. Since the running program and the monitor are a single process now, the time needed to execute the monitoring code can significantly influence the otherwise normal execution of the monitored program.

10.6 Conclusion

Two efficient algorithms for monitoring safety requirements expressed using past time linear temporal logic were presented, one based on rewriting and implemented in Maude, and the other based on dynamic programming, synthesizing specialized monitors from formulae. They both check that a finite sequence of events emitted by a running program satisfies a formula. Operators convenient for monitoring were considered and shown equivalent to standard past time temporal operators.

These algorithms have been implemented in PathExplorer, a runtime verification tool currently under development. The synthesis algorithm has also been implemented (as a plug-in) in the Java-MoP tool [35], which is a general framework for supporting program monitoring for user provided logics; and in the JMPaX tool [163], which extends part of this work to partial order models instead of simple traces.

It is our intention to investigate how the presented algorithms can be refined to work for a logic that combines past and future time temporal logic and that can refer to real-time and data values. Other kinds of runtime verification are also investigated, such as, for example, techniques for detecting error potentials in multi-threaded programs. Recent work on detecting high-level data races is described in [12].

A number of experiments have been carried out with PathExplorer on a planetary rover application written in 35,000 lines of C++. The experiments range from concurrency analysis (deadlock and data race analysis) to monitoring of temporal logic formulae combined with test case generation, as described in [13]. A model checker is used to generate test cases, where a test case consists of input to the application plus a set of temporal formulae that the execution of the application on that input must satisfy. When running this testing environment, hundreds of test cases are generated and the execution of these are monitored against the generated formulae. Initial

experiments have been made with a logic that combines past time and future time temporal logic and supports real-time and data reasoning. A bug was detected in the rover application in the very first such experiment we made. A thread did not detect a premature termination of a certain task in a timely manner. The programmer had forgotten to insert this termination check and was reminded by a single run of the testing environment. This testing environment is planned to become part of the rover application programmer’s testing toolbox.

new stuff below

10.7 Optimal Monitoring of “Always Past” Temporal Safety

A monitor synthesis algorithm from linear temporal logic (LTL) safety formulae of the form $\Box\varphi$ where φ is a past time LTL formula was presented in [87]. The generated monitors implemented the recursive semantics of past-time LTL using a dynamic programming technique, and needed $O(|\varphi|)$ time to process each new event and $O(|\varphi|)$ total space. Some compiler-like optimizations of the generated monitors were also proposed in [87], which would further reduce the required space. It is not clear how much the required space could be reduced by applying those optimizations.

We here show how to generate using a divide-and-conquer technique directly monitors that need $O(k)$ space and still $O(|\varphi|)$ time, where k is the number of temporal operators in φ .

10.7.1 The Monitor Synthesis Algorithm

For simplicity, we assume only two past operators, namely \circ (previously) and \mathcal{S} (since). Let us first note that one cannot asymptotically reduce the space requirements below $\Omega(k)$, where k is the number of temporal operators appearing in the formula to monitor φ . Indeed, one can take $\varphi = (\#_1 \rightarrow t_1) \wedge \cdots \wedge (\#_k \rightarrow t_k)$, where for each $1 \leq i \leq k$, $\#_i$ is some event and t_i is some temporal formula containing precisely one past temporal operator, i.e., a \circ or a \mathcal{S} . Any monitor for φ must directly or indirectly store the status of each t_i at every event, to be able to react accordingly in case the next event is some $\#_i$. Assuming that the events $\#_i$ are distinct

and that the formulae t_i are unrelated, then the monitor needs to distinguish among 2^k possible states, so it needs $\Omega(k)$ space.

In what follows, we assume the usual recursive semantics of LTL, also presented below, restricted to safety formulae of the form $\Box\varphi$, where φ is a past-time LTL. We adopt the simplifying assumption that the empty trace invalidates any atomic proposition and any past temporal operator; as argued in [87], this may not always be the best choice, but other semantic variations regarding the empty trace present no difficulties for monitoring.

Definition 45 (adapted from [129]) *LTL formulae of the form $\Box\varphi$ (read “always φ ”), where φ is a past-time LTL formula, are called LTL safety formulae; we may call them just safety formulae when LTL is understood from the context. An infinite trace $u \in \Sigma^\omega$ satisfies $\Box\varphi$, written $u \models \Box\varphi$, iff each $w \in \text{prefixes}(u)$ satisfies the past-time LTL formula φ , written also $w \models \varphi$ and defined inductively as follows:*

$w \models \text{true}$	<i>is always true,</i>
$ws \models a$	<i>iff $a(s)$ holds,</i>
$w \models \neg\varphi$	<i>iff $w \not\models \varphi$,</i>
$w \models \varphi_1 \wedge \varphi_2$	<i>iff $w \models \varphi_1$ and $w \models \varphi_2$,</i>
$ws \models \circ\varphi$	<i>iff $w \models \varphi$,</i>
$ws \models \varphi_1 \mathcal{S} \varphi_2$	<i>iff $ws \models \varphi_2$ or $ws \models F\varphi$ and $w \models \varphi_1 \mathcal{S} \varphi_2$</i>
$\epsilon \models \varphi$	<i>is false otherwise</i>

Given safety formula $\Box\varphi$, we let $\mathcal{L}(\Box\varphi) \subseteq \Sigma^\omega$ be the set $\{u \in \Sigma^\omega \mid u \models \Box\varphi\}$.

Proposition 16 $\mathcal{L}(\Box\varphi) \in \text{Safety}^\omega$ for any past-time LTL formula φ .

Proof: By the definition of $\mathcal{L}(\Box\varphi)$ in Definition 45 and the definition of $\Box P$ in Definition 12, one can easily note that $\mathcal{L}(\Box\varphi) = \Box\mathcal{L}(\varphi)$, where $\mathcal{L}(\varphi) = \{w \in \Sigma^* \mid w \models \varphi\}$. Therefore, $\mathcal{L}(\Box\varphi) \in \text{Safety}_\Box^\omega$. The rest follows by Theorem 5. \square

Let us next investigate the problem of monitoring safety properties $P \in \text{Safety}^\omega$ expressed as languages of safety formulae, that is, $P = \mathcal{L}(\Box\varphi)$ for some past-time LTL formula φ . Because of the recursive nature of the satisfaction relation, a first important observation is that the generated monitor only needs to store information regarding the status of temporal operators from the previous state. More precisely, the monitor needs one bit per temporal operator, keeping the satisfaction status of the subformula corresponding to that temporal operator; when a new state is received,

the satisfaction status of the subformula is recalculated according to the recursive semantics above and then the bit is updated. The order in which the temporal operators are processed when a new state is received is important: the nested operators must be processed first.

We next present the actual monitor synthesis algorithm at a high-level. We refrain from giving detailed pseudocode as we did in [87], because different applications may choose different implementation paradigms. For example, we are currently using rewriting techniques to implement the monitor synthesis algorithms in MOP [37]; Section 10.7.2 shows our complete Maude rewriting implementation of the subsequent monitor synthesis algorithm.

Step 1 Let $\varphi_1, \dots, \varphi_k$ be the k subformulae of φ corresponding to temporal operators, such that, if φ_i is a subformula of φ_j , then $i < j$; this can be easily achieved by a DFS traversal of φ .

Step 2 Let $bit[1..k]$ be a vector of k bits initialized to 0 (or false); $bit[i]$ will store information related to φ_i from the previous state:

- if $\varphi_i = \odot\psi$ then $bit[i]$ says if ψ was satisfied at the previous state;
- if $\varphi_i = \psi \mathcal{S} \psi'$ then $bit[i]$ says if φ_i was satisfied at the previous step.

Step 3 Let $bit'[1..k]$ be another vector of k bits; this will be used to store temporary results, which will be moved eventually into the vector $bit[1..k]$.

Step 4 Generate a loop that executes whenever a new state s is available; the body of the loop executes the following code:

Step 4.1 For each i from 1 to k execute a bit assignment as follows, where for a subformula ψ of φ , $\overline{\psi}$ is the boolean expression replacing in ψ each non-nested temporal subformula φ_j by $bit[j]$ if φ_j is a “previously” formula or by $bit'[j]$ if φ_j is a “since” formula, and each remaining atomic proposition a by its satisfaction in the current state, $a(s)$:

- if $\varphi_i = \odot\psi$ then generate the assignment $bit'[i] := \overline{\psi}$
- if $\varphi_i = \psi \mathcal{S} \psi'$ then generate the assignment $bit'[i] := \overline{\psi'} \vee \overline{\psi} \wedge bit[i]$

Step 4.2 Generate the conditional: if $\overline{\varphi}$ is false then error (formula violated)

Step 4.3 Generate code to move the contents of $bit'[1..k]$ into $bit[1..k]$.

Note that the generated monitors are well-defined, because each time a $\bar{\psi}$ boolean expression is generated, all the bits in $bit'[1..k]$ that are needed are already calculated. One can also perform boolean simplifications when calculating $\bar{\psi}$ to reduce runtime overhead even further. For example, in our implementation that also generated the code below (see Section 10.7.2), we used the simplification $\neg\neg\psi = \psi$. To illustrate the monitor generation algorithm above, let us consider the past time LTL formula: $\varphi = \neg(a \wedge \neg(\odot b \wedge (c \mathcal{S} (d \wedge (\neg e \mathcal{S} f))))$. Step 1 produces the following enumeration of φ 's subformulae: $\varphi_1 = \odot b$, $\varphi_2 = \neg e \mathcal{S} f$, and $\varphi_3 = c \mathcal{S} (d \wedge (\neg e \mathcal{S} f))$. The other steps eventually generate the code:

```

bit[1..3] := false;      // three global bits
foreach new state s do {
  // first update the bits in a consistent order
  bit'[1] := b(s);
  bit'[2] := f(s)  $\vee$  ( $\neg e(s) \wedge bit[2]$ );
  bit'[3] := d(s)  $\wedge bit'[2]$   $\vee$  (c(s)  $\wedge bit[3]$ );
  // then check whether the formula is violated
  if a(s)  $\wedge \neg(bit[1] \wedge bit'[3])$  then Error;
  // finally, update the state of the monitor
  bit[1..3] := bit'[1..3]
}
```

It is easy to see that for any past LTL formula φ of k temporal operators, the state of the generated monitor is encoded on k bits, namely the vector $bit[1..k]$. The runtime of the generated monitor is still $O(|\varphi|)$, because each temporal operator in φ results in an assignment and a read operation in the monitor, while each boolean operator in φ is “executed” by the monitor.

10.7.2 A Maude Implementation of the Monitor Synthesizer

We here show a term rewriting implementation of the algorithm above, using the Maude system [46]. Implementations in other languages are obviously also possible; however, rewriting proved to be an elegant means to generate monitors from logical formulae in several other contexts, and so seems to be here. In what follows we show the complete Maude code that takes as input a formula, parses it, generates the monitor, and then pretty prints it. We use the K technique here [152], which is a rewriting-based language and/or

logic definitional technique; to use K, one needs to first upload the generic, i.e., application-independent, module discussed at the end of this section.

Atomic Predicates

We start by defining the atomic state predicates that one can use in formulae. These can be either identifiers (of the form 'a, 'abc, 'a123, etc.; these are provided by the Maude builtin module QID):

```
fmod PREDICATE is
  --- atomic predicates can be quoted identifiers
  protecting QID .
  sort Predicate .
  subsort Qid < Predicate .
endfm
```

Syntax of Formulae

Let us next define the syntax of formulae. We here use Maude's mixfix notation for defining syntax as algebraic operators, where underscores stay for arguments. Also, note that operators are assigned precedences (declared as operator attributes), to relive the user from writing parentheses (the lower the precedence the tighter the binding):

```
fmod SYNTAX is
  protecting PREDICATE .
  sort Formula .
  subsort Predicate < Formula .
  op !_ : Formula -> Formula [prec 20] .
  op _/\_ : Formula Formula -> Formula [prec 23] .
  op !_ : Formula -> Formula [prec 21] .
  op !_ : Formula Formula -> Formula [prec 22] .
endfm
```

Target Language

We are done with the input language. Let us now define the output language. We need a very simple language for implementing the generated monitors, namely one with limited assignment, conditional and looping. The generated code, as well as the target language, play no role in this paper; one is expected to change the language below to one's desired target language (Java, C, C#, assembler, etc.). Our chosen language below has bits, expressions, statements and code. Bits are also expressions; code is a list of statements

composed sequentially using “;” or just concatenation. The syntax below is also making use of precedence attributes. The **format** attributes are useful solely for pretty-printing reasons (see Maude’s manual [46] for details on formatting):

```
fmod CODE is
  --- syntax for the generated code
  protecting PREDICATE + INT + STRING .
  sorts Bit Exp Statement Code .
  subsorts Bit < Exp .
  subsort Statement < Code .
  ops (bit[_]) (bit'[_]) : Nat -> Bit .
  ops (bit[1 .. _]) (bit'[1 .. _]) : Int -> Bit .
  op _(s) : Predicate -> Exp [prec 0] .
  ops true false : -> Exp .
  op !_ : Exp -> Exp [prec 20] .
  op _/\_ : Exp Exp -> Exp [prec 23] .
  op _\/_ : Exp Exp -> Exp [prec 24] .
  op _:=_ : Exp Exp -> Statement [prec 27 format(ni d d d)] .
  op if_then_ : Exp Statement -> Statement
    [format(ni d d ++ --) prec 30] .
  op foreach new state s do _ : Code -> Statement
    [format(n d d d s++ --n)] .
  op Error : -> Statement [format(ni d)] .
  op //_ : String -> Statement [format(ni d d)] .
  op nil : -> Code .
  op _;_ : Code Code -> Code [assoc id: nil prec 40] .
  op __ : Code Code -> Code [assoc id: nil prec 40] .
  op {_} : Code -> Statement [format(d d --ni ++)] .
  --- code simplification rules
  var B : Exp .
  eq !! B = B .
endfm
```

The following module defines the actual monitor synthesis algorithm. We use the K definitional technique here, because it yields a very compact implementation. K is centered on the basic intuition of *computation*; computations are encoded as first-order data-structures that “evolve”, via rewriting, to *results*. Computations are sequentialized using the list constructor “ $_ \rightarrow _$ ”; thus, if K and K’ are computations, then $K \rightarrow K'$ is the computation consisting of K followed by K’. Computations may eventually yield results; for example, $K \rightarrow K'$ may rewrite (in context) to $R \rightarrow K'$, meaning that R is the result that K reduces to. An important feature of K is that one can schedule lists of tasks for reduction; for example, $[K1, K2, K3] \rightarrow K$ may eventually reduce to $[R1, R2, R3] \rightarrow K$, where R1, R2, and R3 are the results

that $K1$, $K2$, and $K3$ reduce to, in this order. To use K , one needs to import the module K discussed at the end of this section. The equations of the module K (three in total) are all about reducing a list of computations to a list of results, supposing that one knows how to reduce one computation to one result.

K is a definitional framework that is generic in computations and results. More precisely, it provides sorts `KComputation` and `KResult`, and expects its user to define the desired computations and results, as well as rules to reduce a computation to a result. Computations typically can be reduced to results only in context; to facilitate this, K provides a sort `KConfiguration`, which is also supposed to be populated accordingly. The sort `KConfiguration` is a multi-set sort over a sort `KConfigurationItem`, where the multi-set constructor is just concatenation; also, the sort `KComputation` is a list sort over `KComputationItem`, where the list constructor is `_->_`. To make use of K , one needs to first define constructors for the sorts `KConfigurationItem`, `KComputationItem` and `KResult`, and then to define how each computation item reduces to a result.

In our case, the computations are the formulae or subformulae that still need to be processed, and the results are the corresponding boolean expressions that need to be checked in the current (generated code) context to see whether the formula has been violated or not. We define the following additional constructors: we add four constructors for configurations, namely “`k`” that wraps the current computation, “`code`” that wraps the current generated code, and “`nextBit`” that wraps the next available bit; we add one main constructor for computations, “`form`”, that wraps a formula, and one constant computation item per operator in the input language (the later is needed to know how to combine back the results of the corresponding subexpressions; finally, we add one constructor for results, “`exp`”, that wraps a boolean expression.

The formula is processed in a depth-first-order, following a divide-and-conquer philosophy. Each subformula is decomposed into a list of computation subtasks consisting of its subformulae, then the corresponding results are composed back into a result corresponding to the original subformula. Recall that equations/rules apply wherever they match, not only at the top. Let us only discuss the two equations defining the “since” (`_S_`), the last two in the module below. The first one is straightforward: it decomposes the task of processing `F1 S F2` to the subtasks of processing `F1` and `F2`; the computation item `S` is placed in the computation structure to prepare

the terrain for the next equation. The next equation applies after F1 and F2 have been processed, say to expressions B1 and B2, respectively; if C is the code generated so far and if I+1 is the next bit available, then the boolean expression corresponding to the current since formula is indeed $\text{bit}'(I+1)$, provided that one adds the corresponding code capturing the recursive semantics of since to the generated code.

```
fmod MONITOR-GENERATION is
  protecting K + SYNTAX + CODE .
  op k : KComputation -> KConfigurationItem .
  op code : Code -> KConfigurationItem .
  op nextBit : Nat -> KConfigurationItem .
  op process : Formula -> KConfiguration .
  op form : Formula -> KComputationItem .
  op exp : Exp -> KResult .
  ops ! /\ 0 S : -> KComputationItem .
  var P : Predicate . vars F F1 F2 : Formula . var C : Code .
  var I : Nat . vars B B1 B2 : Exp . var K : KComputation .
  eq process(F) = k(form(F)) code(nil) nextBit(0) .
  eq k(form(P) -> K) = k(exp(P(s)) -> K) .
  eq form(! F) = form(F) -> ! .
  eq exp(B) -> ! = exp(! B) .
  eq form(F1 /\ F2) = [form(F1),form(F2)] -> /\ .
  eq [exp(B1),exp(B2)] -> /\ = exp(B1 /\ B2) .
  eq form(0 F) = form(F) -> 0 .
  eq k(exp(B) -> 0 -> K) code(C) nextBit(I)
    = k(exp(bit[I + 1]) -> K) code(C ; bit'[I + 1] := B)
    nextBit(I + 1) .
  eq form(F1 S F2) = [form(F1), form(F2)] -> S .
  eq k([exp(B1),exp(B2)] -> S -> K) code(C) nextBit(I)
    = k(exp(bit'[I + 1]) -> K)
    code(C ; bit'[I + 1] := B2 \/ B1 /\ bit[I + 1]) nextBit(I + 1) .
endfm
```

Putting It All together

The following module plugs the code generated above into the general pattern:

```
fmod PRETTY-PRINT is
  protecting MONITOR-GENERATION .
  sort Monitor .
  op genMonitor : Formula -> Code .
  op makeMonitor : KConfiguration -> Code .

  var F : Formula . var B : Exp . var C : Code . vars N M : Nat .
```

```

eq genMonitor(F) = makeMonitor(process(F)) .
eq makeMonitor(k(exp(B)) code(C) nextBit(N))
= bit[1 .. N] := false ;
  foreach new state s do {
    // "first update the bits in a consistent order"
    C ;
    // "then check whether the formula is violated"
    if !(B) then Error ;
    // "finally, update the state of the monitor"
    bit[1 .. N] := bit'[1 .. N]
  } .
endfm

```

Our implementation of the monitor synthesizer is now complete. To use it, one can ask Maude reduce terms of the form `genMonitor(F)`, where `F` is the formula that one wants to generate into a monitor. For example:

```

reduce genMonitor(
  !('a /\ !(0 'b /\ 'c S ('d /\ (! 'e S 'f))))
) .

```

For the formula above, Maude will give the expected answer, pretty printed as follows:

```

\|||||/
--- Welcome to Maude ---
/|||||
Maude 2.2 built: Mar 15 2006 16:37:22
Copyright 1997-2005 SRI International
Sat Jan 27 12:01:20 2007
Maude> in p
=====
fmod K
=====
fmod PREDICATE
=====
fmod SYNTAX
=====
fmod CODE
=====
fmod MONITOR-GENERATION
=====
fmod PRETTY-PRINT
=====
reduce in PRETTY-PRINT :
  genMonitor(! ('a /\ !(0 'b /\ 'c S ('d /\ (! 'e S 'f)))) .

```

```

rewrites: 46 in -93406740ms cpu (1ms real) (~ rewrites/second)
result Code:
bit[1 .. 3] := false ;
foreach new state s do {
  // "first update the bits in a consistent order"
  bit'[1] := 'b(s) ;
  bit'[2] := 'f(s) \/\ ! 'e(s) /\ bit[2] ;
  bit'[3] := 'd(s) /\ bit'[2] \/\ 'c(s) /\ bit[3] ;
  // "then check whether the formula is violated"
  if 'a(s) /\ ! (bit[1] /\ bit'[3]) then
    Error ;
  // "finally, update the state of the monitor"
  bit[1 .. 3] := bit'[1 .. 3]
}

Maude>

```

The K Module

One should upload the next module whenever one wants to use the K technique to define a language, logic or tool. Note that the module below has nothing to do with our particular logic under consideration in this paper; that is the reason for which we exiled it here.

```

fmod K is
  sorts KConfigurationItem KConfiguration .
  subsort KConfigurationItem < KConfiguration .
  op empty : -> KConfiguration .
  op _ : KConfiguration KConfiguration -> KConfiguration [assoc comm id: empty] .

  sorts KComputationItem KNeComputation KComputation .
  subsort KComputationItem < KNeComputation < KComputation .
  op nil : -> KComputation .
  op _->_ : KComputation KComputation -> KComputation [assoc id: nil] .
  op _->_ : KNeComputation KNeComputation -> KNeComputation [ditto] .

  sort KComputationList .
  subsort KComputation < KComputationList .
  op nil : -> KComputationList .
  op _ , _ : KComputationList KComputationList -> KComputationList [assoc id: nil] .

  sort KResult KResultList .
  subsorts KResult < KResultList < KComputation .
  op nil : -> KResultList .
  op _ , _ : KResultList KResultList -> KResultList [assoc id: nil] .

```

```

op [_] : KComputationList -> KComputationItem .
op [_] : KResultList -> KComputationItem .
op [_|_] : KComputationList KResultList -> KComputationItem .

var K : KNeComputation . var Kl : KComputationList .
var R : KResult . var Rl : KResultList .
eq [K,Kl] = K -> [Kl | nil] .
eq R -> [K,Kl | Rl] = K -> [Kl | Rl,R] .
eq R -> [nil | Rl] = [Rl,R] .
endfm

```

To use K, after importing the module above, one should define one's own constructors for configuration items (sort `KConfigurationItem`), for computation items (sort `KComputationItem`), and for results (sort `KResult`). For our example, we defined all these at the beginning of the module `MONITOR-GENERATION`.

Chapter 11

Monitoring “Always-Past” Temporal Safety with Call-Return

Material from [155]

Abstract: We present an extension of past time LTL with call/return atoms, called PTCARET, together with a monitor synthesis algorithm for it. PTCARET includes abstract variants of past temporal operators, which can express properties over traces in which terminated function or procedure executions are abstracted away into a call and a corresponding return. This way, PTCARET can express safety properties about procedural programs which cannot be expressed using conventional linear temporal logics. The generated monitors contain both a local state and a stack. The local state is encoded on as many bits as concrete temporal operators the original formula has. The stack pushes/pops bit vectors of size the number of abstract temporal operators the original formula has: push on begins, pop on ends of procedure executions. An optimized implementation is also discussed and is available to download.

11.1 Introduction

Theoretically speaking, it appears to be straightforward to monitor properties expressed as past time linear temporal logic (PTLTL) formulae, since the fix-point semantics of the temporal operators gives a direct deterministic automaton. The practical challenge in monitoring PTLTL formulae stays in how to do it *efficiently*, both time-wise and memory-wise, so that the added runtime overhead to the observed system is minimal. Since in a real-life runtime verification application there could be millions of monitor instances living at the same time, each observing tens of millions of events (see, e.g., [6, 16] and [39, 40] for numbers and evaluations of runtime verification systems on large benchmarks), every bit of memory or monitor processing time may translate into significantly higher runtime overhead, to an extent that the overall use of runtime verification in a particular application may become unfeasible. For example, in many cases it may not be a good idea to generate an actual deterministic automaton as a monitor, because that may have an exponential or worse size; instead, a non-deterministic automaton performing an NFA-to-DFA construction on the fly saving space exponentially may be more appropriate, or even a monitor that does not store any automaton at all, but has an efficient way to generate the next state on-the-fly.

Havelund and Roşu proposed a monitor synthesis algorithm for past-time LTL (PTLTL) formulae φ [87]. The generated monitors implement the recursive semantics of PTLTL using a dynamic programming technique, and need $O(|\varphi|)$ time to process each new event and $O(|\varphi|)$ total space. Roşu proposed an improved monitor synthesis algorithm for PTLTL in [148] (unpublished technical report) which, using a divide-and-conquer strategy, generates monitors that need $O(k)$ space and still $O(|\varphi|)$ time, where k is the number of temporal operators in φ .

Alur *et al.* gave an extension of linear temporal logic (LTL) with calls and returns [8], called CARET. Unlike LTL, CARET allows for matching call/return states in linear traces, allowing to express program trace properties not expressible using plain LTL. In particular, one can express properties on the execution stack of a program, such as “function g is always called from within function f ”, or structured-programming safety policies such as “each method must release before it terminates all the locks that it acquired during its execution”, or even properties that are allowed to be temporarily violated, such as “user u never directly accesses the passwords file (but may access it through system procedures)”. Because of allowing such important and

desirable safety properties to be formally stated at the same time faithfully including LTL, CARET can be a more attractive temporal logic than LTL, provided of course that the complexity of checking programs against CARET formulae does not make it unfeasible.

We define a past time variant of CARET, called PTCARET, show by examples its usefulness in expressing a series of safety properties involving calls of functions/procedures, and then propose a monitor synthesis algorithm for properties expressed as PTCARET formulae. Motivated by practical reasons, PTCARET distinguishes call/return states from begin/end states: the former take place in the caller’s context, while the latter take place in the callee’s. This simple and standard distinction allows more flexibility and elegance in expressing properties, but requires an additional (but reasonable) constraint on traces: calls always immediately precede begins, and ends always immediately precede returns.

PTCARET conservatively extends PTLTL by adding abstract variants of temporal operators, namely “abstract previously” and “abstract since”. The semantics of these operators is that of their corresponding core PTLTL operators “previously” and “since”, but on the *abstract* trace obtained by collapsing executed functions or procedures into only two states, namely the caller’s state at the call of the invoked function or procedure and the caller’s state at its corresponding return. In other words, from the point of view of the abstract temporal operators, the intermediate states generated during function executions are invisible. Of course, the standard temporal operators continue to “see” the whole trace.

The monitors generated from PTCARET formulae using the proposed algorithm have both a monitor state and a monitor stack, so they can be regarded as push-down automata; however, both the monitor states and the data pushed onto stacks are calculated online, on a by-need basis. The monitor state is encoded on as many bits as standard past time operators in the original formula, while the monitor stack pushes/pops as many bits of data as abstract temporal operators in the original formula. If no abstract temporal operators are used in a PTCARET formula, that is, if the PTCARET formula is a PTLTL formula, then its generated monitor is identical to that obtained using the technique in [148]. In other words, not only is PTCARET a conservative extension of PTLTL, but the proposed monitor synthesis algorithm conservatively extends the best known, provably optimal monitor synthesis algorithm for PTLTL.

The proposed PTCARET monitor synthesis algorithm has been imple-

mented and is available to download and experiment with via a web interface at [15]. The rest of the paper is structured as follows: Section 11.2 discusses PTCARET as an extension of PTLTL; Section 13.1.1 introduces useful derived operators and shows some examples of PTCARET specifications. Section 11.4.2 discusses our monitor synthesis algorithm, including its implementation. Section 11.5 concludes the paper.

11.2 PTLTL and PTCARET

We here recall past time linear temporal logic (PTLTL) and define its extension PTCARET. For simplicity, we assume only two types of past operators, namely “previously” and “since”. Other common or less common temporal operators can be added as derived operators. PTLTL contains only the usual, standard variants of temporal operators, while PTCARET contains both standard and abstract variants. We follow the usual recursive semantics of past time LTL and adopt the simplifying assumption that the empty trace invalidates any atomic proposition and any past temporal operator; as argued in [87], this may not always be the best choice, but other semantic variations regarding the empty trace present no difficulties for monitoring and can easily be accommodated.

Definition 46 *Syntactically, PTLTL consists of formulae over the grammar*

$$\varphi ::= \text{true} \mid a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \odot\varphi \mid \varphi \mathcal{S} \varphi,$$

where a ranges over a set A of state predicates. Other common syntactic constructs can be defined as derived operators in a standard way: *false* is $\neg\text{true}$, $\diamond\varphi$ (“eventually in the past”) is $\text{true} \mathcal{S} \varphi$, $\Box\varphi$ (“always in the past”) is $\neg(\diamond\neg\varphi)$, etc.

LTL’s models, even for its safety fragment, traditionally are *infinite traces* (see, e.g., [129]), where a trace is a sequence of *states*, where a state is commonly abstracted as a set of atomic predicates in A .

refer to Chapter 3 below; or merge some of the text earlier in the book; or even remove it altogether.

According to Lamport [125], a *safety property* is a set of such infinite traces (properties are commonly identified with the sets of traces satisfying them)

such that once an execution “violates” it then it can never satisfy it again later. Formally, a set of infinite traces Q is a safety property if and only if for any infinite trace u , if $u \notin Q$ then there is some finite prefix w of u such that $wv \notin Q$ for all infinite traces v .

It can be shown that there are as many safety properties as real numbers [148]. Unfortunately, any logical formalism can define syntactically only as many formulae as natural numbers. Thus, any logical formalism can only express a small portion of safety properties. In LTL, a common way to specify safety properties is as “always past” formulae, that is, as formulae of the form $\Box\varphi$ (\Box is “always in the future”), where φ is a formula in PTLTL. There are two problems with identifying the problem of monitoring a PTLTL specification φ with checking the running system against the LTL safety formula $\Box\varphi$: on the one hand, LTL has an infinite trace semantics, while during monitoring we only have a finite number of past states available, and, on the other hand, once the LTL formula $\Box\varphi$ is violated then it can never be satisfied in the future. However, a major use of monitoring is in the context of recoverable systems, in the sense that the monitor can trigger recovery code when φ is violated, in the hope that φ will be satisfied from here on. For these reasons, we adopt a slightly modified semantics of past time LTL, namely the one on finite traces borrowed from [87]:

Definition 47 *A (program) state is a set of atomic predicates in A ; let s, s' , etc., denote states, and let $ProgState$ denote the set of all states. A trace is a finite sequence of states in $ProgState^*$; let w, w' , etc., denote traces, and ϵ denote the empty trace. If $w \neq \epsilon$, that is, if $w = w's$ for some trace w' and some state s , then we let $prefix(w)$ denote the trace w' and call it the (concrete) prefix of w , and let $last(w)$ denote the state s . The satisfaction relation $w \models \varphi$ between a trace w and a PTLTL formula φ is defined recursively as follows:*

$w \models true$		<i>is always true,</i>
$w \models a$	<i>iff</i>	$w \neq \epsilon$ and $a \in last(w)$,
$w \models \neg\psi$	<i>iff</i>	$w \not\models \psi$,
$w \models \psi \wedge \psi'$	<i>iff</i>	$w \models \psi$ and $w \models \psi'$,
$w \models \odot\psi$	<i>iff</i>	$w \neq \epsilon$ and $prefix(w) \models \psi$,
$w \models \psi \mathcal{S} \psi'$	<i>iff</i>	$w \neq \epsilon$ and ($w \models \psi'$ or $w \models \psi$ and $prefix(w) \models \psi \mathcal{S} \psi'$).

Unlike other places in the book, previously is defined with the “strong” meaning above.

We next introduce PTCARET as an extension of PTLTL. Syntactically, it only adds abstract versions of the two temporal operators “previously” and “since” to PTLTL; semantically, some special atomic predicates corresponding to calls, returns, begins and ends of functions/procedures need to be assumed, as well as some natural and practically reasonable restrictions on traces.

Definition 48 PTCARET *syntactically extends* PTLTL *with:*

$$\begin{array}{lcl} \varphi & ::= & \dots \\ & | & \overline{\varphi} \\ & | & \varphi \mathcal{S} \varphi \end{array}$$

The former is called “abstract previously” and the latter “abstract since”.

The semantics of abstract previously and since are defined exactly as the semantics of their concrete counterparts, but on an abstract version of the trace from which all the intermediate states of the terminated function or procedure executions are erased. In order for this erasure, or abstraction, process to work, we need to impose some constraints on traces that are always satisfied in practice.

Definition 49 In PTCARET, the set of atomic predicates A contains four special predicates: *call*, *begin*, *end*, and *return*. A state contains at most one of these and is called *call*, *begin*, *end*, or *return* state if it contains the corresponding predicate. PTCARET traces are constrained to the following restrictions:

- (1) any call state, except when the last one, must be immediately followed by a begin state, and any begin state must be immediately preceded by a call state;
- (2) any end state, except when the last one, must be immediately followed by a return state, and any return state must be immediately preceded by an end.

For a trace w as above, we let \overline{w} denote its abstraction, which is obtained by iteratively erasing contiguous subtraces $s_b w' s_e$ of w in which s_b is a begin state, s_e is an end state which is not the last one in w , and w' contains no begin or end states. One more restriction is imposed on PTCARET traces:

- (3) the abstractions of PTCARET traces contain no return states which are not immediately preceded by call states.

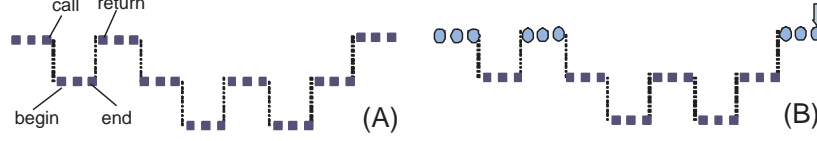


Figure 11.1: PTCARET trace (A) and abstraction (B). \Downarrow : end of w , \blacksquare : w state, \circ : \bar{w} state.

Call and return states occur in the caller's context. Thus, call/return states can contain other predicates which may not be possible to evaluate in the callee's context during runtime monitoring. The begin/end states are generated in the callee's context, at the beginning and at the end of the execution of the invoked function, respectively. Similarly, for some common programming languages, begin/end states may contain other predicates that cannot be evaluated in the caller's context. The original CARET logic [8] did not distinguish between call and begin states or between end and return states. We included all four of them in PTCARET for the reasons above and also because most trace monitoring systems (e.g., Tracematches [6, 16] and MOP [39, 40]) make a clear distinction between these four types of states.

Fig. 11.1 (A) shows a PTCARET trace. To better reflect the call-return structure of the PTCARET trace, states are placed on different levels: states on the higher level are generated in the caller's context while those on the lower level are generated in the callee's. The vertical dotted lines connect the corresponding call-begin and end-return pairs. Fig. 11.1 (B) shows the abstraction of that trace: if w ends with the state pointed by \Downarrow , \bar{w} contains only the circled states.

Restrictions (1) and (2) on PTCARET traces are very natural. One source of doubt though can be the sub-requirements that any return state must be preceded by an end state, and that any begin state must be preceded by a call state. While a return or a begin can indeed happen in any programming language only after a corresponding end or call state, respectively, one may argue that monitoring of a property should be allowed to start at any moment, in particular in between call and begin, or in between end and return states. While our synthesized monitors from PTCARET formulae (see Section 11.4.2) can be easily adapted to start monitoring at any moment in the trace, for the sake of a smoother and simpler development of the theoretical foundations of PTCARET, we assume that any PTCARET trace

starts from the beginning of the program execution and thus satisfies the above-mentioned restrictions. Restriction (3) ensures that a trace does not contain return states that do not have corresponding matching call states, also a natural restriction on complete traces.

Our definition of trace abstraction above is admittedly operational, but we think that it captures the desired end/begin matching concept both compactly and intuitively. Alternatively, we could have followed the CARET style in [8] and define the matching begin state of an end state as the latest begin state containing a balanced number of begin/end states in between.

uncomment the next text?

Definition 50 For a non-empty PTCARET trace w , let $\overline{\text{prefix}}(w)$, called the *abstract prefix* of w (not to be confused with the abstraction of the prefix of w , $\text{prefix}(w)$), be either $\text{prefix}(w)$ if $\text{last}(w)$ is not a return state, or otherwise the prefix of w up to and including the corresponding matching call state of $\text{last}(w)$ if it is a return state; formally, if $\text{last}(w)$ is a return state then $\overline{\text{prefix}}(w)$ is the trace $w's_c$, where $w = w'w''$ for some w'' with $\overline{w''} = s_cs_r$, where s_c and s_r are call and return states, respectively.

Fig. 11.2 illustrates $\overline{\text{prefix}}(w)$ on two traces, with the down arrow pointing to the ends of the traces. In Fig. 11.2 (A) we assume that w ends with a state that is not a **return** (the arrow points to a call state) and in Fig. 11.2 (B) w ends with a **return** state (the states of the corresponding $\overline{\text{prefix}}(w)$ are marked with diamonds).

Definition 51 The satisfaction relation between a PTCARET trace w and a PTCARET formula φ is defined recursively exactly like in PTLTL for the PTLTL operators, and as follows for the two abstract temporal operators:

$$\begin{aligned} w \models \overline{\circ}\psi & \quad \text{iff} \quad w \neq \epsilon \text{ and } \overline{\text{prefix}}(w) \models \psi, \\ w \models \psi \overline{\mathcal{S}} \psi' & \quad \text{iff} \quad w \neq \epsilon \text{ and } (w \models \psi' \text{ or } w \models \psi \text{ and } \overline{\text{prefix}}(w) \models \psi \overline{\mathcal{S}} \psi'). \end{aligned}$$

Therefore, a formula $\overline{\circ}\psi$ is satisfied in a return state iff ψ was satisfied at the corresponding matching call state. It is satisfied in a non-return state, including an end state, iff $\circ\psi$ is satisfied in that state (that is, if and only if ψ was satisfied in the concrete (non-abstract) previous state).

Fig. 11.3 compares the \circ and $\overline{\circ}$ operators. The arrows point, for each state, where the formula ψ in $\circ\psi$ (A) and in $\overline{\circ}\psi$ (B) holds. For most states,

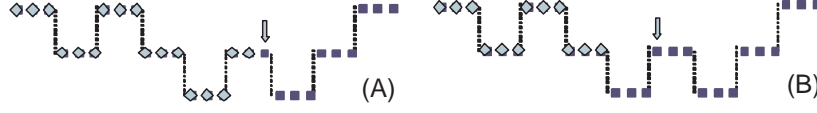


Figure 11.2: $\overline{prefix}(w)$ on two traces, (A) and (B). \Downarrow : the end of w , \diamond : state in $\overline{prefix}(w)$.

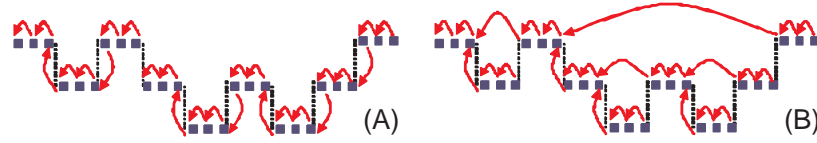


Figure 11.3: Concrete (A) and abstract (B) “previous” states for \odot and $\overline{\odot}$.

their abstract previous state is the concrete previous one; the only difference is on return states, because the abstract previous state of a return state is its call state.

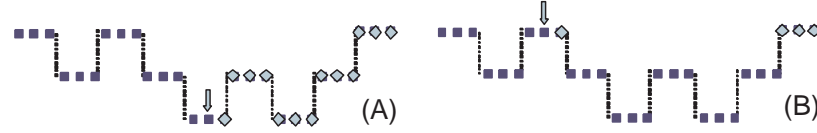


Figure 11.4: $\psi \mathcal{S} \psi'$ (A) versus $\psi \overline{\mathcal{S}} \psi'$ (B). \Downarrow : where ψ' holds, \diamond : where ψ holds.

Figure 11.4 compares $\psi \mathcal{S} \psi'$ and $\psi \overline{\mathcal{S}} \psi'$. Notice that the various call/return levels play no role in the satisfaction of $\psi \mathcal{S} \psi'$, but that they play a crucial role in the satisfaction of $\psi \overline{\mathcal{S}} \psi'$: for the latter, ψ' must hold on the same level or a higher level as the level of the current state. One can show the following expected property of abstract since:

Proposition 17 $\varphi_1 \overline{\mathcal{S}} \varphi_2$ is semantically equivalent to $\varphi_2 \vee \varphi_1 \wedge \overline{\odot}(\varphi_1 \overline{\mathcal{S}} \varphi_2)$.

One should not get tricked and assume that $w \models \overline{\odot} \varphi$ if and only if $\overline{w} \models \odot \varphi$, or that $w \models \varphi_1 \overline{\mathcal{S}} \varphi_2$ if and only if $\overline{w} \models \varphi_1 \mathcal{S} \varphi_2$! The reason is that subformulae φ , φ_1 or φ_2 may contain concrete temporal operators whose

semantics still involve the entire execution trace, not only the abstract one. Some examples in this category are shown in Section 13.1.1. Nevertheless, the following holds:

Proposition 18 *For a PTCARET trace w and formula φ containing no concrete temporal operators \odot and \mathcal{S} , $w \models \varphi$ iff $\bar{w} \models \hat{\varphi}$, where $\hat{\varphi}$ is the PTLTL formula replacing each abstract temporal operator in φ by its concrete variant.*

11.3 PTCARET Derived Operators and Examples

Besides the usual derived Boolean operators and past time temporal operators “eventually in the past”, “always in the past”, as well as “start”, “stop”, and “interval” operators like in [87], which can all be also defined abstract variants, we can define several other interesting, PTCARET-specific derived operators. In the rest of the paper we use the standard notation for the derived Boolean operators, e.g., “ \rightarrow ”, “ \vee ”, etc., with their usual precedences, and assume that “ \odot ” binds as tight as “ \neg ” while “ \mathcal{S} ” binds tighter than the binary Boolean operators.

At beginning. Suppose that one would like a particular property, say ψ , to hold at the beginning of the execution of the current function. We can define the derived temporal operator $@_b$, say “at beginning”, as follows:

$$@_b\psi \stackrel{\text{def}}{=} (\text{begin} \rightarrow \psi) \wedge (\neg\text{begin} \rightarrow \odot(\text{begin} \rightarrow \psi)\overline{\mathcal{S}}\text{begin}).$$

Note that the concrete “previously” operator is used inside the argument of the “abstract since” operator. The above is correct because the last begin state seen by the “abstract since” is indeed the beginning of the current function or procedure. One should not get tricked and try to define the above as:

$$@_b\psi \stackrel{\text{def}}{=} (\text{begin} \rightarrow \psi) \wedge (\neg\text{begin} \rightarrow (\text{begin} \rightarrow \psi)\overline{\mathcal{S}}\text{call}).$$

That is because the current function may have called and returned from several other functions, and the “abstract since” can still see all the call/return states. The above would vacuously hold in such a case.

At call. Suppose now that one wants ψ to hold at the state when the current function was called. For the same reason as above, one cannot simply replace **begin** by **call** in the definition of $@_b$ above. However, one

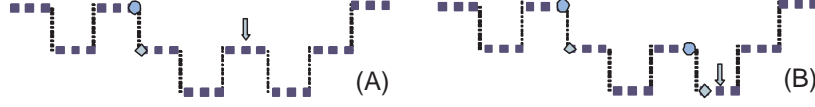


Figure 11.5: Derived operators. \Downarrow : current state, \diamond : states for $@_b$, \mathcal{S}_b ; \circ : states for $@_c$, \mathcal{S}_c

can define the derived temporal operator $@_c$, say “at call”, in terms of “at beginning” simply as follows:

$$@_c\psi \stackrel{\text{def}}{=} @_b\psi.$$

In Fig. 11.5 (A), supposing that the current state is the one pointed to by the arrow, ψ should hold in the diamond state for $@_b\psi$ and in the circle state for $@_c\psi$.

Stack since on beginnings. The “abstract since” can be used to write properties in which the terminated function executions are irrelevant. There may be cases in which one wants to write properties referring exclusively to the execution stack of a program, ignoring any other states. For example, one may want to say that ψ held on the stack since property ψ' held. As usual, one may be interested in properties ψ and ψ' to hold either at call time, or at execution beginning time. Let us first define a “stack since on beginnings” derived operator:

$$\psi \overline{\mathcal{S}_b} \psi' \stackrel{\text{def}}{=} (\text{begin} \rightarrow \psi) \overline{\mathcal{S}} (\text{begin} \wedge \psi').$$

Stack since on calls. To define a “stack since on calls” one cannot simply replace **begin** by **call** in the above. Instead, one can define it as follows:

$$\varphi_1 \overline{\mathcal{S}_c} \varphi_2 \stackrel{\text{def}}{=} (\text{call} \rightarrow \varphi_1) \overline{\mathcal{S}} (\text{begin} \wedge \circ \varphi_2).$$

In Fig. 11.5 (B), if the current state is the one pointed by the arrow, the begin stack consists of the diamonds and the call stack consists of the circles.

With the stack since derived temporal operators above, one can further define other derived operators, such as “stack eventually in the past on calls” (say $\overline{\diamond_c}$), “stack always in the past on beginnings” (say $\overline{\Box_b}$), etc.

Let us next further illustrate the strength of PTCARET by specifying some concrete properties that would be hard or impossible to specify in PTLTL.

Suppose that in a particular context, function f must be called only directly by function g . Assuming $call_f$ and $call_g$ are predicates that hold when f and g are called, respectively, we can specify this property in PTCARET as follows:

$$call_f \rightarrow @_c call_g.$$

Suppose now that f can be called only directly or indirectly by g : a call to g must be on the stack whenever f is called. We can specify that as follows:

$$call_f \rightarrow \overline{\diamond}_c call_g.$$

A common safety property in many systems is that resources acquired during a function execution must be released before the function ends. Assuming that *acquire* and *release* are predicates that hold when the resource of interest is acquired or released, respectively, we can specify this property as follows:

$$\text{end} \rightarrow (\neg \text{acquire} \overline{\mathcal{S}} \text{begin} \vee \neg(\neg \text{release} \overline{\mathcal{S}} \text{acquire})).$$

A more complex example is discussed in Section 11.4.3.

11.4 A Monitor Synthesis Algorithm for PTCARET

As discussed in [148] for PTLTL, thanks to the recursive nature of the satisfaction relation on the standard PTLTL temporal operators (see Definition 47), the monitor generated from a PTCARET formula needs only one global bit per standard (non-abstract) temporal operator. This bit maintains the satisfaction status of the subformula corresponding to that standard temporal operator; when a new state is observed, the satisfaction status of that subformula is recalculated according to the recursive semantics in Definition 47 and the bit is updated. In order for this to work, one needs to have already updated or have an easy way to calculate the status of the subformulae.

The situation is more complex for the abstract temporal operators, as one needs to store enough information about the past so that one is able to

update the status of abstract operators' satisfaction regardless of how the future evolves. The main complication comes from the fact that one needs to “freeze” the satisfaction status of the subformulae corresponding to abstract temporal operators whenever a begin state is observed, and then “unfreeze” it when the corresponding end state is observed, thus recovering the information that was available right when the function call took place. Fortunately, that can be obtained by using a stack to push/pop the satisfaction status of the abstract temporal subformulae.

More precisely, a stack bit is needed per abstract temporal operator in the PTCARET formula, maintaining the satisfaction status of the subformula corresponding to that abstract operator. When a new state is observed, the satisfaction status of that subformula is recalculated according to the recursive semantics in Definition 50 and the stack bit updated; if the newly observed state is a begin, then the status of the stack bits is pushed on the stack *before* the actual monitor state update; if the newly observed state is an end, then the status of the stack bits is popped from the stack *after* the monitor state update.

11.4.1 The Target Language

To state and prove the correctness of any program generation algorithm, one needs to have a formal semantics of the target language. This section gives a formal syntax and semantics to the simple and generic language in which we synthesize monitors. One can very easily translate this language into standard languages, such as C, C++, C#, Java, or even into native machine code. For each PTCARET formula φ , we are going to generate (in Section 11.4.2) a monitor \mathcal{M}_φ as a statement in a language \mathcal{L}_φ . The only difference between the languages \mathcal{L}_φ is the set of variables that one can assign values to; the rest of the language constructs are the same for all φ . The language \mathcal{L}_φ has the following simple syntax (note that $\mathcal{L}_{\varphi_1} \subseteq \mathcal{L}_{\varphi_2}$ whenever φ_1 is a subformula of φ_2):

$$\begin{array}{ll}
Var & ::= \alpha_\phi \text{ (one for each subformula } \phi \text{ of } \varphi \text{ rooted in } \odot \text{ or } \mathcal{S}) \\
& \quad | \beta_\phi \text{ (one for each subformula } \phi \text{ of } \varphi \text{ rooted in } \odot \text{ or } \overline{\mathcal{S}}) \\
Exp & ::= true \mid A \mid Var \mid \neg Exp \mid Exp \wedge Exp \\
Stm & ::= Var := Exp \mid \text{if begin then push} \mid \text{if end then pop} \\
& \quad | \text{output}(Exp) \mid Stm \ Stm
\end{array}$$

Therefore, programs in \mathcal{L}_φ can use predicates in A (the atomic predicate set of PTCARET) as ordinary (Boolean) expressions, together with Boolean variables α_ϕ and β_ϕ , one per standard and abstract temporal operator in φ , respectively, and together with Boolean constructs such as complement and conjunction. Statements can be composed using juxtaposition, and can be: α_ϕ or β_ϕ variable assignment, output of a Boolean expression, or conditional push/pop, the latter pushing or popping, by convention, precisely the bit vector $\vec{\beta}$. We assume a (rather conventional) denotational semantics for \mathcal{L}_φ as follows:

Definition 52 *If φ has k_1 standard temporal operators and k_2 abstract temporal operators, then let $MonState_\varphi$ (we think of \mathcal{L}_φ programs as monitors) be the state space of \mathcal{L}_φ , that is, the domain $Bool^{k_1} \times Bool^{k_2} \times Stack \times Output$, where $Bool$ is the set $\{true, false\}$, $Stack$ is the domain $(Bool^{k_2})^*$ of stacks, or lists, over bit vectors of size k_2 , and $Output$ is the domain $Bool^*$ of bit lists. Let the functions*

$$\begin{aligned} \llbracket - \rrbracket : Exp &\rightarrow MonState_\varphi \rightarrow ProgState \rightarrow Bool \\ \llbracket - \rrbracket : Stm &\rightarrow MonState_\varphi \rightarrow ProgState \rightarrow MonState_\varphi \end{aligned}$$

be defined as follows:

$$\begin{aligned} \llbracket true \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= true, \quad \llbracket a \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) = s(a), \\ \llbracket \alpha_\phi \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= \vec{\alpha}(i), \text{ where } i \leq k_1 \text{ is the } \vec{\alpha}\text{-index corresponding to } \phi, \\ \llbracket \beta_\phi \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= \vec{\beta}(j), \text{ where } j \leq k_2 \text{ is the } \vec{\beta}\text{-index corresponding to } \phi, \\ \llbracket b_1 \wedge b_2 \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= \llbracket b_1 \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) \text{ and } \llbracket b_2 \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s), \\ \llbracket \alpha_\phi := b \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= (\vec{\alpha}[\vec{\alpha}(i) \leftarrow \llbracket b \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s)], \vec{\beta}, \sigma, \omega), \\ \llbracket \beta_\phi := b \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= (\vec{\alpha}, \vec{\beta}[\vec{\beta}(j) \leftarrow \llbracket b \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s)], \sigma, \omega), \\ \llbracket \text{if begin then push} \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= \begin{cases} (\vec{\alpha}, \vec{\beta}, \vec{\beta} \cdot \sigma, \omega) & \text{if } s(\text{begin}), \\ (\vec{\alpha}, \vec{\beta}, \sigma, \omega) & \text{otherwise,} \end{cases} \\ \llbracket \text{if end then pop} \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= \begin{cases} (\vec{\alpha}, \vec{\beta}', \sigma', \omega) & \text{if } s(\text{end}) \text{ and } \sigma = \vec{\beta}' \cdot \sigma', \\ (\vec{\alpha}, \vec{\beta}, \sigma, \omega) & \text{otherwise,} \end{cases} \\ \llbracket output(b) \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= (\vec{\alpha}, \vec{\beta}, \sigma, \omega \cdot \llbracket b \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)), \\ \llbracket stm \text{ } stm' \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= \llbracket stm' \rrbracket(\llbracket stm \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s)). \end{aligned}$$

We can now associate a function $\llbracket \mathcal{M}_\varphi \rrbracket : MonState_\varphi \rightarrow ProgState \rightarrow MonState_\varphi$ to each program \mathcal{M}_φ in \mathcal{L}_φ . For a monitor state $(\vec{\alpha}, \vec{\beta}, \sigma, \omega) \in MonState_\varphi$ and a program state $s \in ProgState$, $\llbracket \mathcal{M}_\varphi \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) = (\vec{\alpha}', \vec{\beta}', \sigma', \omega')$

if and only if the monitor \mathcal{M}_φ executed in state $(\vec{\alpha}, \vec{\beta}, \sigma, \omega)$ when program state s is observed, produces monitor state $(\vec{\alpha}', \vec{\beta}', \sigma', \omega')$.

Definition 53 *By abuse of notation, we also let $\llbracket \mathcal{M}_\varphi \rrbracket : \text{ProgState}^* \rightarrow \text{MonState}_\varphi$ be the function (false^{k_1} is the vector of k_1 false bits, and ϵ is the empty list):*

$$\begin{cases} \llbracket \mathcal{M}_\varphi \rrbracket(\epsilon) = (\text{false}^{k_1}, \text{false}^{k_2}, \epsilon, \epsilon) & \text{— the “initial” monitor state —} \\ \llbracket \mathcal{M}_\varphi \rrbracket(ws) = \llbracket \mathcal{M}_\varphi \rrbracket(\llbracket \mathcal{M}_\varphi \rrbracket(w))(s) \end{cases}$$

11.4.2 The Monitor Synthesis Algorithm

We next present the actual monitor synthesis algorithm at a high-level. We refrain from giving detailed pseudocode as in [87], because different applications may choose different implementation paradigms. For example, our implementation of the PTCARET logic plugin in the context of the MOP system [39, 40], discussed in Section 11.4.3, uses term rewriting techniques. The monitoring code for a PTCARET formula φ can be split into three pieces: code to be executed before the monitor outputs the satisfaction status of the formula, the outputting code, and code to be executed after the output. Let $\text{Code}_{\text{before}}^\varphi$ denote the former and let $\text{Code}_{\text{after}}^\varphi$ denote the latter.

$\text{Code}_{\text{before}}^\varphi$ is concerned with updating the status of the “since” operators in a bottom-up fashion, while $\text{Code}_{\text{after}}^\varphi$ with updating the status of the “previously” operators. Indeed, in order to output the satisfaction status of φ , one needs to know the status of all the “since” operators, which may depend upon values in the current state as well as upon values of nested “since” operators, so the inner “since” operators need to be processed before the outer ones. On the other hand, one need not know the particular details (values of atomic predicates) of the current state in order to know the status of the “previously” operators; all one needs to make sure of is that the status of the “previously” operators has been updated at the appropriate previous state (or states in the case of “abstract previously”), after the monitor output. Interestingly, note that, unlike the “since” operators, the “previously” operators need to be processed in a top-down fashion, that is, the outer ones need to be processed before the inner ones.

Note that the monitors \mathcal{M}_φ generated in Figure 11.6 are well-defined, in the sense that each time a generated Boolean expression $\bar{\psi}$ is executed, all the α and β bits that are needed have been calculated. That is because

the code is generated following a DFS traversal of the original `PTCARET` formula. \mathcal{M}_φ is run at each newly generated event, or program state, and outputs either *true* or *false*. Note that each \mathcal{M}_φ has the form “(if **begin** then push) C_1^φ **output**(O^φ) C_2^φ (if **end** then pop)”, for some potential statements C_1^φ and C_2^φ , and for some Boolean expression O^φ . To simplify notation, we introduce the following:

Definition 54 Let $\langle C_1^\varphi, O^\varphi, C_2^\varphi \rangle$ be a shorthand for:

if begin then push
 C_1^φ
output(O^φ)
 C_2^φ
if end then pop

We use \emptyset for C_1^φ or C_2^φ when they do not exist.

The following result structurally relates monitors generated for formulae φ to monitors generated for its subformulae. One can use this proposition as an equivalent, recursive way to synthesize monitors for `PTCARET`:

Proposition 19 If $\mathcal{M}_\psi = \langle C_1^\psi, O^\psi, C_2^\psi \rangle$ and $\mathcal{M}_{\psi'} = \langle C_1^{\psi'}, O^{\psi'}, C_2^{\psi'} \rangle$ then:

- $\mathcal{M}_{true} = \langle \emptyset, true, \emptyset \rangle$
- $\mathcal{M}_a = \langle \emptyset, a, \emptyset \rangle$
- $\mathcal{M}_{\neg\psi} = \langle C_1^\psi, \neg O^\psi, C_2^\psi \rangle$
- $\mathcal{M}_{\psi \wedge \psi'} = \langle C_1^\psi \ C_1^{\psi'}, O^\psi \wedge O^{\psi'}, C_2^{\psi'} \ C_2^\psi \rangle$
- $\mathcal{M}_{\odot\psi} = \langle C_1^\psi, \alpha_{\odot\psi}, (\alpha_{\odot\psi} := \overline{\psi}) \ C_2^\psi \rangle$
- $\mathcal{M}_{\psi \mathcal{S} \psi'} = \langle C_1^\psi \ C_1^{\psi'} \ (\alpha_{\psi \mathcal{S} \psi'} := \overline{\psi'} \vee \overline{\psi} \wedge \alpha_{\psi \mathcal{S} \psi'}), \alpha_{\psi \mathcal{S} \psi'}, C_2^{\psi'} \ C_2^\psi \rangle$
- $\mathcal{M}_{\overline{\odot}\psi} = \langle C_1^\psi, \beta_{\overline{\odot}\psi}, (\beta_{\overline{\odot}\psi} := \overline{\psi}) \ C_2^\psi \rangle$
- $\mathcal{M}_{\psi \overline{\mathcal{S}} \psi'} = \langle C_1^\psi \ C_1^{\psi'} \ (\beta_{\psi \overline{\mathcal{S}} \psi'} := \overline{\psi'} \vee \overline{\psi} \wedge \beta_{\psi \overline{\mathcal{S}} \psi'}), \beta_{\psi \overline{\mathcal{S}} \psi'}, C_2^{\psi'} \ C_2^\psi \rangle$

To prove the correctness of our monitor synthesis algorithm, we need to show that after observing any sequence of program states w , a synthesized monitor \mathcal{M}_φ outputs the same result as the satisfaction status of $w \models \varphi$. Therefore, we need to define “the output of the monitor \mathcal{M}_φ after observing w ”:

Definition 55 Let $\llbracket \mathcal{M}_\varphi \rrbracket : \text{ProgState}^+ \rightarrow \text{Bool}$ be defined for each (non-empty) $w \in \text{ProgState}^+$ as $\llbracket \mathcal{M}_\varphi \rrbracket(w) = b$ iff $\llbracket \mathcal{M}_\varphi \rrbracket(w) = (\vec{\alpha}, \vec{\beta}, \sigma, \omega \cdot b)$. For uniformity, let us extend $\llbracket \mathcal{M}_\varphi \rrbracket$ to a function $\text{ProgState}^* \rightarrow \text{Bool}$ (as in Definitions 47 and 50):

- $\llbracket \mathcal{M}_\varphi \rrbracket(\epsilon) = \text{false}$ when $\varphi = a, \odot\psi, \psi \mathcal{S} \psi', \overline{\odot}\psi, \psi \overline{\mathcal{S}} \psi'$;
- $\llbracket \mathcal{M}_{\neg\psi} \rrbracket(\epsilon) = \neg \llbracket \mathcal{M}_\psi \rrbracket(\epsilon)$;
- $\llbracket \mathcal{M}_{\psi \wedge \psi'} \rrbracket(\epsilon) = \llbracket \mathcal{M}_\psi \rrbracket(\epsilon) \wedge \llbracket \mathcal{M}_{\psi'} \rrbracket(\epsilon)$.

Proposition 20 The following hold for any $w \in \text{ProgState}^*$:

- $\llbracket \mathcal{M}_{\text{true}} \rrbracket(w)$ is always true,
- $\llbracket \mathcal{M}_a \rrbracket(w)$ iff $w \neq \epsilon$ and $a \in \text{last}(w)$,
- $\llbracket \mathcal{M}_{\neg\psi} \rrbracket(w)$ iff not $\llbracket \mathcal{M}_\psi \rrbracket(w)$,
- $\llbracket \mathcal{M}_{\psi \wedge \psi'} \rrbracket(w)$ iff $\llbracket \mathcal{M}_\psi \rrbracket(w)$ and $\llbracket \mathcal{M}_{\psi'} \rrbracket(w)$,
- $\llbracket \mathcal{M}_{\odot\psi} \rrbracket(w)$ iff $w \neq \epsilon$ and $\llbracket \mathcal{M}_\psi \rrbracket(\text{prefix}(w))$,
- $\llbracket \mathcal{M}_{\psi \mathcal{S} \psi'} \rrbracket(w)$ iff $w \neq \epsilon$ and ($\llbracket \mathcal{M}_{\psi'} \rrbracket(w)$ or $\llbracket \mathcal{M}_\psi \rrbracket(w)$ and $\llbracket \mathcal{M}_{\psi \mathcal{S} \psi'} \rrbracket(\text{prefix}(w))$),
- $\llbracket \mathcal{M}_{\overline{\odot}\psi} \rrbracket(w)$ iff $w \neq \epsilon$ and $\llbracket \mathcal{M}_\psi \rrbracket(\overline{\text{prefix}}(w))$,
- $\llbracket \mathcal{M}_{\psi \overline{\mathcal{S}} \psi'} \rrbracket(w)$ iff $w \neq \epsilon$ and ($\llbracket \mathcal{M}_{\psi'} \rrbracket(w)$ or $\llbracket \mathcal{M}_\psi \rrbracket(w)$ and $\llbracket \mathcal{M}_{\psi \overline{\mathcal{S}} \psi'} \rrbracket(\overline{\text{prefix}}(w))$).

Proof: The non-trivial ones are those for temporal operators. We only discuss $\overline{\mathcal{S}}$, because the others follow the same idea and are simpler. The monitors for $\psi \overline{\mathcal{S}} \psi'$, ψ , and ψ' , respectively, following the notations in Proposition 19 are:

	$\mathcal{M}_{\psi \overline{\mathcal{S}} \psi'}$	\mathcal{M}_ψ	$\mathcal{M}_{\psi'}$
1.	if begin then push	if begin then push	if begin then push
2.	$C_1^\psi \ C_1^{\psi'}$	C_1^ψ	$C_2^{\psi'}$
3.	$\beta_{\psi \mathcal{S} \psi'} := \overline{\psi'} \vee \overline{\psi} \wedge \beta_{\psi \mathcal{S} \psi'}$		
4.	output($\beta_{\psi \mathcal{S} \psi'}$)	output($\overline{\psi}$)	output($\overline{\psi'}$)
5.	$C_2^{\psi'} \ C_2^\psi$	C_2^ψ	$C_2^{\psi'}$
6.	if end then pop	if end then pop	if end then pop

Note that the property holds vacuously if $w = \epsilon$. Assume now that $w = w's$, for some $s \in \text{ProgState}$. An interesting and useful property of the generated

monitors is that their semantics is very modular, and that pushing or popping $\vec{\beta}$ does not affect the modular semantics. For example, note that C_1^ψ in $\mathcal{M}_{\psi \mathcal{S} \psi'}$ uses no variables defined in $C_1^{\psi'}$ or in $C_2^{\psi'}$, and the bit $\beta_{\psi \mathcal{S} \psi'}$ is only defined in line 3. and used in lines 3. and 4. This modularity guarantees that, if we were to output $\bar{\psi}$ or $\bar{\psi}'$ at line 3. or 4. in $\mathcal{M}_{\psi \mathcal{S} \psi'}$, then its output after processing w would be nothing but $\llbracket \mathcal{M}_\psi \rrbracket(w)$ or $\llbracket \mathcal{M}_{\psi'} \rrbracket(w)$, respectively. That means that the $\bar{\psi}$ and $\bar{\psi}'$ in the expression assigned to $\beta_{\psi \mathcal{S} \psi'}$ at line 4. when processing the last state in w are $\llbracket \mathcal{M}_\psi \rrbracket(w)$ and $\llbracket \mathcal{M}_{\psi'} \rrbracket(w)$, respectively. We claim that $\beta_{\psi \mathcal{S} \psi'}$ in the assigned expression at line 4. is $\llbracket \mathcal{M}_{\psi \mathcal{S} \psi'} \rrbracket(\overline{\text{prefix}(w)})$. There are two cases to analyze. (1) if s is not a return state, then $\beta_{\psi \mathcal{S} \psi'}$ was assigned at line 3. in the previous execution of the monitor, when processing the last state in w' , so it is nothing but $\llbracket \mathcal{M}_{\psi \mathcal{S} \psi'} \rrbracket(\text{prefix}(w))$; and (2) if s is a return state, then it means that the last state in w' was an end state, so the vector $\vec{\beta}$ was popped from the stack at the end of the previous step. The only thing left to note is that our **push** on begins and **pop** or ends correctly match **begin** and **end** states; this follows from the fact that we assume traces complete and well-formed (Definition 49). \square

Theorem 20 *The monitor synthesis algorithm in Figure 11.6 is correct, that is, for any PTCARET formula φ and for any $w \in \text{ProgState}^*$, $\llbracket \mathcal{M}_\varphi \rrbracket(w)$ iff $w \models \varphi$.*

Proof: Straightforward, by induction on both the structure of φ and the length of w , noticing that there is a one-to-one correspondence between the definition of satisfaction in Definitions 47 and 50, and the properties in Proposition 20. \square

11.4.3 Implementation as Logic Plugin, Optimizations, Example

MOP [39, 40] is a configurable runtime verification framework, in which specification requirements formalisms can be added modularly, by means of *logic plugins*. A logic plugin essentially encapsulates a monitor synthesis algorithm for a formalism that one can then use to specify properties of programs. The current JavaMOP tool has logic plugins for future time LTL, past time LTL, Allen algebra, extended regular expressions, JML, JASS. JavaMOP takes a Java application to be monitored and specifications using any of the included formalisms together with validation and/or violation

handlers (saying what to do if property validated or violated, in particular nothing), and then waves them together in a runtime verified application by first generating monitors for all the properties using their corresponding logic plugins, and then generating and compiling an AspectJ extension of the original program (runtime monitors are “aspects”). To maintain a reduced runtime overhead (shown on large benchmarks to be, on average, below 10%), MOP piggybacks monitor states onto object states.

The PTCARET MOP logic plugin. We implemented the PTCARET monitor synthesis algorithm in Section 11.4.2 as an MOP logic plugin. Our implementation can be found and experimented with online at [15]. Large-scale experiments are still to be performed; we are currently engineering the MOP system to allow monitor states to piggyback not only object states, but also the program stack. In short, our implementation uses *term rewriting* and the Maude system [46], and follows the monitor synthesis algorithm in Figure 11.6 and its “equivalent”, recursive formulation in Proposition 19. Implementations in other languages are obviously also possible; however, term rewriting proved to be an elegant means to synthesize monitors from logical formulae in several other contexts (the other MOP plugins, as well as in JPaX [150]), and so seems to be here.

Our implementation starts by defining the Boolean expressions as an algebraic specification using Maude’s mixfix notation (equivalent to context-free grammars); derived Boolean operators are also defined, together with several simplification rules ($\neg \text{true} = \text{false}$, etc.). Boolean expressions are imported both in the target language module and in the PTCARET module. Both the target language and the PTCARET modules are defined as algebraic signatures, enriched with structural equalities which turn into simplification rules when executed; this way, for example, each PTCARET derived operator is defined with one equation capturing its definition. Several other derived operators are defined in addition to those discussed in Section 13.1.1. The monitor generation module imports both the target language and the PTCARET modules, and adds two equations per temporal logic operator; e.g., the equations below process the “abstract since”:

```
eq form(F1 Sa F2) = [form(F1), form(F2)] -> Sa .
eq k([exp(B1),exp(B2)] -> Sa -> K) code(I,C1,C2) nextBeta(N)
  = k(exp(beta[N]) -> K) code(I beta[N] := false,
                                C1 beta[N] := B2 or B1 and beta[N], C2) nextBeta(N + 1) .
```

First equation says that subformulae should be processed first (DFS traversal). The second equation combines the codes generated from the subformulae as

shown in Proposition 19, appending the assignment for the corresponding bit to **C1**. Note that **C1** here accumulates the “code before” of both subformulae; in terms of Proposition 19, it is “ $C_1^{\psi} C_1^{\psi'}$ ”. **I** accumulates the monitor initialization code. Finally the optimizations below are implemented also as rewrite rules.

Optimizations. Term-rewriting-based code-generation algorithms can be easily extended with optimizations, because these can be captured as rewrite rules. We discuss some of the optimizations enabled in our implementation. First, we perform Boolean simplifications when calculating $\bar{\psi}$ to reduce runtime overhead ($\neg\neg\psi = \psi$, $\text{true} \wedge \psi = \text{true}$, etc.). Another immediate optimization is the following. The generated code originally has the form (see Fig. 11.6) “(if begin then push) C (if end then pop)”, for some code C . However, since a program state can only contain at most one of the special predicates, this can be optimized into (syntax of target language needs to be slightly extended):

```

if begin then {
    push
     $C[\text{begin} \leftarrow \text{true}, \text{end} \leftarrow \text{false}, \text{call} \leftarrow \text{false}, \text{return} \leftarrow \text{false}]$ 
    exit
}
if end then {
     $C[\text{begin} \leftarrow \text{false}, \text{end} \leftarrow \text{true}, \text{call} \leftarrow \text{false}, \text{return} \leftarrow \text{false}]$ 
    pop
    exit
}
 $C[\text{begin} \leftarrow \text{false}, \text{end} \leftarrow \text{false}]$ 

```

After the substitutions above, further Boolean simplifications may be triggered. Also, some assignments may become redundant, such as, for example, “**beta**[3] := **beta**[3]”; rules to eliminate such assignments are also given. A further optimization on the generated code is possible, but we have not implemented it yet: some subformulae can repeat in different parts of the original formula; the current implementation generates monitoring code for each repeating instance, which is redundant and can be reduced using a smarter optimization algorithm.

Example. We here show the monitor generated by our implementation for a more complex PTCARET specification. Suppose that a program carries

out a critical multi-phase task and the following safety properties must hold when execution enters the second phase:

1. Execution entered the first phase within the same procedure;
2. Resource acquired within same procedure since first phase must be released;
3. Caller of current procedure must have had approval for the second phase;
4. Task is executed directly or indirectly by the procedure *safe_exec*.

These can be captured as the following PTCARET formula:

$$\begin{aligned}
enter_phase_2 \rightarrow & \quad (\neg(\neg enter_phase_1 \overline{S} \text{begin}) \\
& \quad \wedge (\neg acquire \overline{S} enter_phase_1 \vee \neg(\neg release \overline{S} acquire)) \\
& \quad \wedge @_c(has_phase_2_pass) \\
& \quad \wedge \overline{\otimes}_b(safe_exec)
\end{aligned}$$

Our implementation generates the following monitor for this specification:

```

if begin then {
  push(beta);
  beta[0] := safe_exec or beta[0];
  beta[1] := enter_ph1 or not acquire and beta[1];
  beta[2] := acquire or not release and beta[2];
  beta[3] := true; beta[4] := true;
  output(not enter_ph2 or not beta[4] and alpha[0] and beta[0] and (not beta[2] or beta[1]));
  alpha[3] := true;
  alpha[2] := alpha[1];
  alpha[1] := has_ph2_pass;
  alpha[0] := has_ph2_pass;
  exit
}
if end then {
  beta[1] := enter_ph1 or not acquire and beta[1];
  beta[2] := acquire or not release and beta[2];
  beta[3] := beta[3] and (not alpha[3] or alpha[2]);
  beta[4] := not enter_ph1 and beta[4];
  output(not enter_ph2 or not beta[4] and beta[0] and beta[3] and (not beta[2] or beta[1]));
  alpha[3] := false;
  alpha[2] := alpha[1];
  alpha[1] := has_ph2_pass;
  alpha[0] := has_ph2_pass;
  pop(beta);
  exit
}
beta[1] := enter_ph1 or not acquire and beta[1];

```

```

beta[2] := acquire or not release and beta[2];
beta[3] := beta[3] and (not alpha[3] or alpha[2]);
beta[4] := not enter_ph1 and beta[4];
output(not enter_ph2 or not beta[4] and beta[0] and beta[3] and (not beta[2] or beta[1]));
alpha[3] := false;
alpha[2] := alpha[1];
alpha[1] := has_ph2_pass;
alpha[0] := has_ph2_pass

```

The formula contains derived operators, e.g., $@_c$, which are first expanded. The monitoring code uses four α bits and five β bits (the expanded formula contains four concrete temporal operators and five abstract ones). For example, $\Diamond_b(\text{safe_exec})$ is expanded into $(\text{begin} \rightarrow \text{true})\overline{\mathcal{S}}(\text{begin} \wedge \text{safe_exec})$, which is then simplified to $\text{true}\overline{\mathcal{S}}(\text{begin} \wedge \text{safe_exec})$, equivalent to $\Diamond(\text{begin} \wedge \text{safe_exec})$. **beta**[0] in the generated code is used to check this operation; it only needs to be updated at the **begin** state, where it becomes true if *safe_exec* holds.

11.5 Conclusion and Future Work

We presented the logic PTCARET and a monitor synthesis algorithm for it. PTCARET includes abstract variants of past temporal operators. It can express safety properties about procedural programs which cannot be expressed using conventional PTLTL. The generated monitors contain both a local state and a stack. The local state is encoded on as many bits as concrete temporal operators the original formula had, while the stack pushes/pops bit vectors of size the number of abstract temporal operators the original formula had. An optimized implementation of the monitor synthesis algorithm has been organized as an MOP logic plugin, and is available to download from [15]. There is room for further optimizations of the generated code. An extensive evaluation of the effectiveness of PTCARET runtime verification on large programs needs to be conducted. On the theoretical side, it would be interesting to explore the relationship between our monitors generated for PTCARET and the nested word automata in [33]; [33] gives an operational monitoring language for nested words based on BLAST's specification language. In contrast, our language is declarative and an operational encoding synthesized automatically.

uncommend the next text

11.6 Auxiliary Material - not included in original paper

In this section we show via an example how to generate dynamic programming code for a concrete *ptLTL*-formula. We think that this example would practically be sufficient for the reader to foresee our general algorithm presented in the next subsection. Let $\uparrow p \rightarrow [q, \downarrow (r \vee s)]_s$ be the *ptLTL*-formula that we want to generate code for. The formula states: “whenever p becomes true, then q has been true in the past, and since then we have not yet seen the end of r or s ”. The code translation depends on an enumeration of the subformulae of the formula that satisfies the *enumeration invariant*: any formula has an enumeration number smaller than the numbers of all its subformulae. Let $\varphi_0, \varphi_1, \dots, \varphi_8$ be such an enumeration:

$$\begin{aligned}\varphi_0 &= \uparrow p \rightarrow [q, \downarrow (r \vee s)]_s, \\ \varphi_1 &= \uparrow p, \\ \varphi_2 &= p, \\ \varphi_3 &= [q, \downarrow (r \vee s)]_s, \\ \varphi_4 &= q, \\ \varphi_5 &= \downarrow (r \vee s), \\ \varphi_6 &= r \vee s, \\ \varphi_7 &= r, \\ \varphi_8 &= s.\end{aligned}$$

Note that the formulae have here been enumerated in a post-order fashion. One could have chosen a breadth-first order, or any other enumeration, as long as the enumeration invariant is true.

The input to the generated program will be a finite trace $t = s_1 s_2 \dots s_n$ of n events. The generated program will maintain a state via a function $update : \mathbf{State} \times Event \rightarrow \mathbf{State}$, which updates the state with a given event.

In order to illustrate the dynamic programming aspect of the solution, one can imagine recursively defining a matrix $s[1..n, 0..8]$ of boolean values $\{0, 1\}$, with the meaning that $s[i, j] = 1$ iff $t_i \models \varphi_j$. Then one can fill the table according to the recursive semantics of past time LTL as described in Subsection 10.3.3. This would be the standard way of regarding the above satisfaction problem as a dynamic programming problem. An important observation is, however, that, like in many other dynamic programming algorithms, one doesn't have to store the entire table $s[1..n, 0..8]$, which would be quite large in practice; in this case, one needs only $s[i, 0..8]$ and

$s[i-1, 0..8]$, which we'll write $now[0..8]$ and $pre[0..8]$ from now on, respectively. It is now only a relatively simple exercise to write up the following algorithm for checking the above formula on a finite trace:

```

State  $state \leftarrow \{\}$ ;

bit  $pre[0..8]$ ;

bit  $now[0..8]$ ;

INPUT: trace  $t = s_1s_2...s_n$ ;

/* Initialization of  $state$  and  $pre$  */

 $state \leftarrow update(state, s_1)$ ;

 $pre[8] \leftarrow s(state)$ ;

 $pre[7] \leftarrow r(state)$ ;

 $pre[6] \leftarrow pre[7] \text{ or } pre[8]$ ;

 $pre[5] \leftarrow \text{false}$ ;

 $pre[4] \leftarrow q(state)$ ;

 $pre[3] \leftarrow pre[4] \text{ and not } pre[5]$ ;

 $pre[2] \leftarrow p(state)$ ;

 $pre[1] \leftarrow \text{false}$ ;

 $pre[0] \leftarrow \text{not } pre[1] \text{ or } pre[3]$ ;

/* Event interpretation loop */

for  $i = 2$  to  $n$  do {

     $state \leftarrow update(state, s_i)$ ;

     $now[8] \leftarrow s(state)$ ;

     $now[7] \leftarrow r(state)$ ;

     $now[6] \leftarrow now[7] \text{ or } now[8]$ ;

```

```

    now[5] ← not now[6] and pre[6];
    now[4] ← q(state);
    now[3] ← (pre[3] or now[4]) and not now[5];
    now[2] ← p(state);
    now[1] ← now[2] and not pre[2];
    now[0] ← not now[1] or now[3];
    if now[0] = 0 then
        output('‘property violated’');
        pre ← now;
    };

```

In the following we explain the generated program.

Declarations Initially a state is declared. This will be updated as the input event list is processed. Next, the two arrays *pre* and *now* are declared. The *pre* array will contain values of all subformulae in the previous state, while *now* will contain the value of all subformulae in the current state.

Initialization The initialization phase consists of initializing the *state* variable and the *pre* array. The first event s_1 of the event list is used to initialize the *state* variable. The *pre* array is initialized by evaluating all subformulae bottom up, starting with highest formula numbers, and assigning these values to the corresponding elements of the *pre* array; hence, for any $i \in \{0 \dots 8\}$ $pre[i]$ is assigned the initial value of formula φ_i . The *pre* array is initialized in such a way as to maintain the view that the initial state is supposed stationary before monitoring is started. This in particular means that $\uparrow p$ is false, as well as is $\downarrow (r \vee s)$, since there is no change in state (indexes 1 and 5). The interval operator has the obvious initial interpretation: the first argument must be true and the second false for the formula to be true (index 3). Propositions are true if they hold in the initial state (indices 2, 4, 7 and 8), and boolean operators are interpreted the standard way (indices 0, 6).

Event Loop The main evaluation loop goes through the event trace, starting from the second event. For each such event, the state is updated, followed by assignments to the *now* array in a bottom-up fashion similar to the initialization of the *pre* array: the array elements are assigned values from higher index values to lower index values, corresponding to the values of the corresponding subformulae. Propositional boolean operators are interpreted the standard way (indices 0 and 6). The formula $\uparrow p$ is true if p is true now and not true in the previous state (index 1). Similarly with the formula $\downarrow (r \vee s)$ (index 5). The formula $[q, \downarrow (r \vee s)]_s$ is true if either the formula was true in the previous state, or q is true in the current state, and in addition $\downarrow (r \vee s)$ is not true in the current state (index 3). At the end of the loop an error message is issued if *now*[0], the value of the whole formula, has the value 0 in the current state. Finally, the entire *now* array is copied into *pre*.

Given a fixed *ptLTL* formula, the analysis of this algorithm is straightforward. Its time complexity is $\Theta(n)$ where n is the length of the input trace, the constant being given by the size of the *ptLTL* formula. The memory required is constant, since the length of the two arrays is the size of the *ptLTL* formula. However, one may want to also include the size of the formula, say m , into the analysis; then the time complexity is obviously $\Theta(n \cdot m)$ while memory required is $2 \cdot (m + 1)$ bits. The authors conjecture that it's hard to find an algorithm running faster than the above in practical situations, though some slight optimizations are possible (see Section 10.5.3).

11.6.1 The Algorithm Formalized

We now formally describe our algorithm that synthesizes a dynamic programming algorithm from a *ptLTL*-formula. It takes as input a formula and generates a program as the one above, containing a “for” loop which traverses the trace of events, while validating or invalidating the formula. The generated program is printed using the function **output**, which takes one or more string or integer parameters which are concatenated in the output. This algorithm is designed to generate pseudocode, but it can easily be adapted to generate code in any imperative programming language:

INPUT: past time LTL formula φ

let $\varphi_0, \varphi_1, \dots, \varphi_m$ be the subformulae of φ ;

```

output("State  $state \leftarrow \{\}$ ");
output("bit  $pre[0..m]$ ");
output("bit  $now[0..m]$ ");
output("INPUT: trace  $t = s_1s_2...s_n$ ");
output("/* Initialization of  $state$  and  $pre$  */");
output("state  $\leftarrow update(state, s_1)$ ");
for  $j = m$  downto 0 do {
    output("     $pre[$ ,  $j$ ,  $]$   $\leftarrow$  ");
    if  $\varphi_j$  is a variable then
        output( $\varphi_j$ , "(state)");
    if  $\varphi_j$  is true then output("true");
    if  $\varphi_j$  is false then output("false");
    if  $\varphi_j = \neg\varphi_{j'}$  then
        output("not  $pre[$ ,  $j'$ ,  $]$ ");
    if  $\varphi_j = \varphi_{j_1} \text{ op } \varphi_{j_2}$  then
        output("pre[ $$ ,  $j_1$ ,  $]$  op pre[ $$ ,  $j_2$ ,  $]$ ");
    if  $\varphi_j = \odot\varphi_{j_1}$  then
        output("pre[ $$ ,  $j_1$ ,  $]$ ");
    if  $\varphi_j = \diamond\varphi_{j_1}$  then
        output("pre[ $$ ,  $j_1$ ,  $]$ ");
    if  $\varphi_j = \Box\varphi_{j_1}$  then
        output("pre[ $$ ,  $j_1$ ,  $]$ ");
    if  $\varphi_j = \varphi_{j_1} S_s \varphi_{j_2}$  then
        output("pre[ $$ ,  $j_2$ ,  $]$ ");

```

```

if  $\varphi_j = \varphi_{j_1} S_w \varphi_{j_2}$  then
    output(“pre[”,  $j_1$ , “] or pre[”,  $j_2$ , “];”);
if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_s$  then
    output(“pre[”,  $j_1$ , “] and not pre[”,  $j_2$ , “];”);
if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_w$  then
    output(“not pre[”,  $j_2$ , “];”);
if  $\varphi_j = \uparrow \varphi_{j'}$  then output(“false;”);
if  $\varphi_j = \downarrow \varphi_{j'}$  then output(“false;”);
};
output(“/* Event interpretation loop */”);
output(“for  $i = 2$  to  $n$  do {”);
for  $j = m$  downto 0 do {
    output(“    now[”,  $j$ , “]  $\leftarrow$  ”);
    if  $\varphi_j$  is a variable then output( $\varphi_j$ , “(state);”);
    if  $\varphi_j$  is true then output(“true;”);
    if  $\varphi_j$  is false then output(“false;”);
    if  $\varphi_j = \neg \varphi_{j'}$  then output(“not now[”,  $j'$ , “];”);
    if  $\varphi_j = \varphi_{j_1} op \varphi_{j_2}$  then
        output(“now[”,  $j_1$ , “] op now[”,  $j_2$ , “];”);
    if  $\varphi_j = \odot \varphi_{j_1}$  then output(“pre[”,  $j_1$ , “];”);
    if  $\varphi_j = \diamond \varphi_{j_1}$  then
        output(“pre[”,  $j$ , “] or now[”,  $j_1$ , “]);”);
    if  $\varphi_j = \Box \varphi_{j_1}$  then
        output(“pre[”,  $j$ , “] and now[”,  $j_1$ , “]);”);

```



```

if  $\varphi_j = \varphi_{j_1} S_s \varphi_{j_2}$  then
  output(“(pre[”, j, “] and now[”, j1, “] or
    now[”, j2, “];”);
if  $\varphi_j = \varphi_{j_1} S_w \varphi_{j_2}$  then
  output(“(pre[”, j, “] and now[”, j1, “] or
    now[”, j2, “];”);
if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_s$  then
  output(“(pre[”, j, “] or now[”, j1, “] and
    not now[”, j2, “];”);
if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_w$  then
  output(“(pre[”, j, “] or now[”, j1, “] and
    not now[”, j2, “];”);
if  $\varphi_j = \uparrow \varphi_{j'}$  then
  output(“now[”, j', “] and
    not pre[”, j', “];”);
if  $\varphi_j = \downarrow \varphi_{j'}$  then
  output(“not now[”, j', “] and
    pre[”, j', “];”);
};
output(“    if now[0] = 0 then
  output(“‘property violated’”);
output(“    pre ← now;”);
output(“}”);

```

op is any binary propositional connective. Since we have already given a detailed explanation of the example in the previous section, we shall only give a very brief description of this algorithm.

The formula should be first visited top down to assign increasing numbers to subformulae as they are visited. Let $\varphi_0, \varphi_1, \dots, \varphi_m$ be the list of all

subformulae. Because of the recursive nature of *ptLTL*, this step ensures us that the truth value of $t_i \models \varphi_j$ can be completely determined from the truth values of $t_i \models \varphi_{j'}$ for all $j < j' \leq m$ and the truth values of $t_{i-1} \models \varphi_{j'}$ for all $j \leq j' \leq m$.

Before we generate the main loop, we should first generate code for initializing the array *pre*[0..*m*], basically giving it the truth values of the subformulae on the initial state, conceptually being an infinite trace with repeated occurrences of the initial state. After that, the generated main event loop will process the events. The loop body will update/calculate the array *now* and in the end will move it into the array *pre* to serve as basis for the next iteration. After each iteration *i*, *now*[0] tells whether the formula is validated by the trace $s_1s_2\dots s_i$.

Since the formula enumeration procedure is linear, the algorithm synthesizes a dynamic programming algorithm from an *ptLTL* formula in linear time with the size of the formula. The boolean operations used above are usually very efficiently implemented on any microprocessor and the arrays of bits *pre* and *now* are small enough to be kept in cache. Moreover, the dependencies between instructions in the generated “for” loop are simple to analyze, so a reasonable compiler can easily unfold or/and parallelize it to take advantage of machine’s resources. Consequently, the generated code is expected to run very fast. We shall next illustrate how such optimizations can be part of the translation algorithm.

INPUT: A PTCARET formula φ

OUTPUT: Code that monitors φ

Step 1 Allocate a bit α_ϕ , initially *false*, for each subformula ϕ of φ rooted in a standard temporal operator. Intuition for this bit is:

- if $\phi = \odot\psi$ then α_ϕ says if ψ was satisfied at the previous state;
- if $\phi = \psi \mathcal{S} \psi'$ then α_ϕ says if ϕ was satisfied at previous state.

Step 2 Allocate a bit β_ϕ , initially *false*, for each subformula ϕ of φ rooted in an abstract temporal operator. Intuition for this bit is:

- if $\phi = \overline{\odot}\psi$ then β_ϕ says if ψ was satisfied at abstract previous state;
- if $\phi = \psi \overline{\mathcal{S}} \psi'$, β_ϕ says if ϕ was satisfied at abstract previous state.

Step 3 Initialize $Code_{before}^\varphi$ and $Code_{after}^\varphi$ as follows:

- $Code_{before}^\varphi$ to the code “if begin then push”, and
- $Code_{after}^\varphi$ to the code “if end then pop”.

Notation: For subformulae ϕ of φ , let $\overline{\phi}$ be the Boolean expression replacing in ϕ each temporal-operator-rooted subformula ψ which is not a subformula of another temporal-operator-rooted subformula of ϕ , by either α_ψ when ψ is rooted in a standard temporal operator, or by β_ψ when ψ is rooted in an abstract operator. E.g., $a \wedge \odot b \overline{\mathcal{S}} c \wedge \odot (d \mathcal{S} \overline{\odot} e)$ is $a \wedge \beta_{\odot b \overline{\mathcal{S}} c} \wedge \alpha_{\odot (d \mathcal{S} \overline{\odot} e)}$.

Step 4 Following a depth-first-search (DFS) traversal of φ , for each subformula ϕ of φ rooted in a temporal operator do:

- if $\phi = \odot\psi$ then $Code_{after}^\varphi \leftarrow (\alpha_\phi := \overline{\psi}) \text{ } Code_{after}^\varphi$
- if $\phi = \overline{\odot}\psi$ then $Code_{after}^\varphi \leftarrow (\beta_\phi := \overline{\psi}) \text{ } Code_{after}^\varphi$
- if $\phi = \psi \mathcal{S} \psi'$ then $Code_{before}^\varphi \leftarrow Code_{before}^\varphi (\alpha_\phi := \overline{\psi'} \vee \overline{\psi} \wedge \alpha_\phi)$
- if $\phi = \psi \overline{\mathcal{S}} \psi'$ then $Code_{before}^\varphi \leftarrow Code_{before}^\varphi (\beta_\phi := \overline{\psi'} \vee \overline{\psi} \wedge \beta_\phi)$

Step 5 Output monitor \mathcal{M}_φ as the code “ $Code_{before}^\varphi \text{ output}(\overline{\varphi}) \text{ } Code_{after}^\varphi$ ”

Figure 11.6: The monitor synthesis algorithm for PTCARET

Chapter 12

Efficient Monitoring of Context-Free Patterns

Chapter 13

Efficient Monitoring with Deterministic String Rewrite Systems

Abstract: Early efforts in runtime verification show that parametric regular and temporal logic specifications can be monitored efficiently. These approaches, however, have limited expressiveness: their specifications always reduce to monitors with finite state. More recent developments showed that parametric context-free properties can be efficiently monitored with overheads generally lower than 12–15%. While context-free grammars are more expressive than finite-state languages, they still do not allow every computable safety property. This paper presents a monitor synthesis algorithm for string rewriting systems (SRS). SRSs are well known to be Turing complete, allowing for the formal specification of any computable safety property. Earlier attempts at Turing complete monitoring have been relatively inefficient. This paper demonstrates that monitoring parametric SRSs is practical. The presented algorithm uses a modified version of Aho-Corasick string searching for quick pattern matching with an incremental rewriting approach that avoids reexamining parts of the string known to contain no redexes.

13.1 Introduction

Runtime verification (RV) is a formal analysis approach in which specifications of requirements are given together with the code to check, as in traditional formal verification, but the code is checked against its requirements at runtime, as in testing. A large number of runtime verification techniques and systems, including TemporalRover [58], JPaX [84], JavaMaC [116], Hawk/Eagle [53], Tracematches [6, 16], J-Lo [24], PQL [133], PTQL [78], MOP [40, 39], Pal [34], RuleR [20], etc., have been developed recently, and the overall approach has gained enough traction to spawn its own conference [19]. In a runtime verification system, monitoring code is generated from the specified properties and integrated with the system to monitor. Therefore, a runtime verification approach consists of at least three interrelated aspects: (1) a specification formalism, used to state properties to monitor, (2) a monitor synthesis algorithm, and (3) a means to instrument programs. The chosen specification formalism determines the expressivity of the runtime verification approach and/or system.

Monitoring safety properties is arbitrarily complex [159]. Early developments in runtime verification, showed that *parametric* regular and temporal logic-based formal specifications can be efficiently monitored against large programs. A parametric monitor associates monitor states with different object instantiations for the given parameters. This allows for specification of properties about the relationships of objects, e.g., a relationship between a Collection object and its associated Iterator objects in Java¹. As shown by experiments with Tracematches [16] and the most recent experiments using JavaMOP [111], parametric regular and temporal logic specifications can be monitored against large programs with little runtime overhead, on the order of 15% or lower.

However, both regular expressions and temporal logics are monitored using finite automata, so they have inherently limited expressivity. More specifically, most runtime verification approaches and systems consider only *flat execution traces*, or execution traces without any structure. Consequently, users of such runtime verification systems are prevented from specifying and checking *structured properties*, those properties referring to the program structure such as properties with requirements on the contents of the program call stack. PQL [133], Hawk/Eagle [53], and RuleR [20] provide more expressive logics, but these are relatively inefficient [6, 16, 40]. More

¹ Typestates [173], a popular concept in software engineering and software analysis, can be monitored with parametric monitors that have only one parameter.

recently, JavaMOP was extended to support efficient context-free monitors with runtime overheads very similar to the earlier finite-state logics [134]. While this work allows for checking many structured properties, it does not have the full power to specify any possible safety property. In this paper, we introduce an algorithm for monitoring parametric deterministic string rewriting systems, to serve as an efficient runtime verification technique for specifying and monitoring arbitrarily complex properties; indeed, string rewriting systems are known to be as expressive as Turing machines [27]. We also provide an implementation of our algorithm as an MOP *logic plugin* [40, 39], so it can be used as an integral part of the JavaMOP runtime verification system. This finally gives JavaMOP the ability to monitor any possible safety property with a formal specification. By abuse of vocabulary, we will refer to deterministic string rewriting systems as string rewriting systems and abbreviate them SRSs.

13.1.1 Examples

The JavaMOP specification presented in Figure 13.1, which uses the new **srs** logic plugin, is able to catch situations in which a monitored program attempts to write to a closed `FileWriter`. JavaMOP specifications begin with a declaration of the name of the specification and parameters. Here the property is named `SAFEFILEWRITER`, and one parameter `f` of type `FileWriter`. The parameters allow us to associate separate monitor states with each object instantiation of the parameters. In this case, with one parameter, there will be one monitor state associated with each object instance of `FileWriter`. This is important because we would not want calls to different object instances of the `FileWriter` class to interfere with each other.

The next part of a JavaMOP specification is the declaration of events. Here we are able to generate three different events: `open`, `write`, and `close`. The events are defined using a superset [135] of AspectJ [114] advice with embedded pointcuts. Here, the event `open` occurs when the `FileWriter` constructor is called.

After the event definitions, we list the formalized property. The keyword **srs** tells JavaMOP that the following property will be a deterministic string rewriting system. Rules in our SRS formalism take the form “ $l \rightarrow r$.”, meaning that the string of events on the left hand side of the arrow rewrites to that on the right side. We only need two rules to specify our desired property. The first rule states that we replace `open write` with `open`. This allows us to collapse multiple safe write operations. The second rule

```

SafeFileWriter(FileWriter f) {

    event open after() returning(FileWriter f) :
        call(FileWriter.new(..)) {}
    event write before(FileWriter f) :
        call(* write(..) && target(f) {})
    event close after(FileWriter f) :
        call(* close(..) && target(f) {})

    srs :
        open write -> open .
        close write -> #fail .

    @fail {
        System.out.println("write after close");
    }
}

```

Figure 13.1: SAFEFILEWRITER SRS SPECIFICATION

catches our misuse case: when we have a write after a close. Here we use one of the three special keywords in the SRS plugin, `#fail`, that signifies that a failure has occurred. Also available is `#succeed`, which allows for denoting success, and `#epsilon`, which simply deletes the left-hand side of a rule from the current string of events.

Note that the SRS rules can be applied in any order when a new event is received, so it is user's responsibility to write *confluent* SRSs or to use the deterministic order of rule application explained in Section 13.3.1. The last part of a JavaMOP specification is the handler section. Handlers are arbitrary Java code that is executed when the monitor raises a particular condition. Here the keyword `@fail` denote that the code within the subsequent braces is run when the string rewrites to `#fail`. Using `@succeed` works respectively for the `#succeed` keyword. The SRS algorithm allows any arbitrary handler keyword other than `#epsilon`. In this example, the handler simply prints out an informative message when an invalid write occurs. In general, handler code may be used for anything, such as running a specific algorithm or recovering from the error denoted by the failure of the safety property in question.

Here we show two further examples of safety policies expressed using

deterministic SRS. We show only the property without worrying about the definition of events or handlers for the sake of brevity. The first property, called HASNEXT, is a property of the Java `Iterator` interface stating that `hasNext()` should always be called and return `true` before `next` is called. Below it is specified as a regular expression:

$$(\text{hasnexttrue next})^* \text{next}$$

The corresponding SRS is as follows:

$$\begin{aligned} \text{hasnexttrue next} &\rightarrow \# \text{epsilon} \\ \text{hasnexttrue hasnexttrue} &\rightarrow \text{hasnexttrue} \\ \text{^next} &\rightarrow \# \text{fail} \end{aligned}$$

While this SRS is certainly larger than the original ERE, it may be easier to understand by some users because it directly captures the semantics of the property by simply enumerating all the cases that one has to worry about. The rule `hasnexttrue hasnexttrue → hasnexttrue` conveys the notion that multiple calls to the `hasNext()` method are idempotent. `hasnexttrue next` rewrites to `#epsilon` because it is a safe operation. If `next` is seen at the beginning of the string a failure is raised as `hasnexttrue` was not properly called. Because our algorithm is incremental and deterministically rewrites from left to right it is not strictly necessary to match the beginning of the string, but it is more clear conceptually.

The second property is called SAFELock which corresponds to the proper nesting of acquiring and releasing locks. Proper nesting, in this case, means that corresponding calls to `acquire()` and `release()` occur within the same method body. Here `begin` and `end` denote the beginning and end of a method body.

$$\begin{aligned} S &\rightarrow \epsilon \mid S \text{ acquire } M \text{ release } A \\ M &\rightarrow \epsilon \mid M \text{ begin } M \text{ end } \mid M \text{ acquire } M \text{ release} \\ A &\rightarrow \epsilon \mid A \text{ begin } \mid A \text{ end} \end{aligned}$$

The property is fairly complex, and a complete explanation can be found in [134]. The SRS for the property follows:

$$\begin{aligned} \text{begin end} &\rightarrow \# \text{epsilon} \\ \text{acquire release} &\rightarrow \# \text{epsilon} \\ \text{begin release} &\rightarrow \# \text{tooManyReleases} \\ \text{acquire end} &\rightarrow \# \text{tooFewReleases} \end{aligned}$$

In this case, the SRS is quite a bit less complex than the context-free grammar specifying the same safety property. Again, it conveys interesting semantic information. From the SRS it is clear that a `begin` followed immediately by a `release()` results in an error because we require all `release()` to occur in the same method call as the corresponding `acquire()`. Similarly, an `acquire()` follow by a `end` results in an error because the lock is not correctly released within the method body. `begin end` and `acquire release` rewrite to `#epsilon` because they are properly nested when they occur adjacently. The SRS is also able to encode information that cannot be encoded in the context-free grammar. By using the handlers `#tooManyReleases` and `#tooFewReleases` we are able to run different handlers when there are too many or too few releases, respectively. This allows us to, for example, ignore an extraneous release or to add a missed release into the control stream. This cannot be done with the context-free grammar without adding extra Java code to the MOP specification.

13.1.2 Contributions

There are two main contributions to this paper:

- An efficient, optimized string rewriting algorithm. It builds upon a modification of the Aho-Corasick algorithm [4]. The original algorithm was designed for quickly finding strings in text. Our modified algorithm keeps track of substitution boundaries so that a rewrite step can be performed in time linear to the length of the right hand side of the matched rule². To our knowledge, this is the first time it has been applied to string rewriting. An optimization has also been devised, which checks for early termination of rewriting.
- An implementation and extensive evaluation of the above algorithm as an MOP logic plugin for runtime verification. This way, it can serve as a specification formalism for parametric safety properties in instances of the MOP framework, such as JavaMOP. We show that its performance in practical runtime verification of large systems is acceptable when compared to other means to specify the same properties. Additionally, we show that it outperforms one of the state-of-the-art rewrite engines, Maude [44], which implicitly supports string rewriting as rewriting modulo associativity.

²The right hand side must be copied, so that the rule is still viable the next time it matches.

13.1.3 Paper Outline

Section 13.2 presents related work in the field of runtime verification: popular runtime monitoring systems, with a particular emphasis on those with greater than finite state specification languages and on our framework of choice for our implementation and experimental test-bed, MOP. Section 13.3 presents our string rewriting algorithm, with its use and construction of pattern match automata and optimization that allows for early termination. Section 13.4 presents our experimental results, and Section 13.5 concludes.

13.2 Related Work and MOP

Many approaches have been proposed to monitor program execution against formally specified properties. Interested readers can refer to [135] for an extensive discussion on existing runtime monitoring approaches. Briefly, all runtime monitoring approaches except MOP have their specification formalisms hardwired, and few of them share the same logic.

There are four orthogonal attributes of a runtime monitoring system: logic, scope, running mode, and handlers. The logic specifies which formalism is used to specify the property. The scope determines where to check the property; it can be class invariant, global, interface, etc. The running mode denotes where the monitoring code runs; it can be inline (weaved into the code), online (operating at the same time as the program), outline (receiving events from the program remotely, e.g., over a socket), or offline (checking logged event traces). The handlers specify what actions to perform under exceptional conditions; such conditions include violation and/or validation of the property. It is worth noting that for some logics, violation and validation are not complementary to each other, i.e., the violation of a formula does not always imply the validation of the negation of the formula. MOP allows for handlers for any number of user defined exceptional situations (called handler categories).

Most runtime monitoring approaches can be framed in terms of these attributes, as illustrated in Figure 13.2, which shows an (incomplete) summary of runtime monitoring systems. For example, JPaX can be regarded as an approach that uses linear temporal logic (LTL) to specify class-scoped properties, whose monitors work in offline mode and only detect violation. In general, JavaMOP (the Java instance of MOP) has proven to be the most efficient of the runtime monitoring systems despite being generic in logical formalism.

Approach	Logic	Scope	Mode	Handler
JPaX [84]	LTL	class	offline	violation
TemporalRover [58]	MiTL	class	inline	violation
Monopoly [21]	MFOTL	global	offline	validation
Larva [50]	DATE (timed automata)	multiple	online	transitions
JavaMaC [116]	PastLTL	class	outline	violation
Hawk [53]	Eagle	global	inline	violation
RuleR [20]	RuleR	global	inline	violation
Tracematches [16]	Reg. Exp.	global	inline	validation
J-Lo [24]	LTL	global	inline	violation
Pal [34]	modified Blast	global	inline	validation
PQL [133]	PQL	global	inline	validation
PTQL [78]	SQL	global	outline	validation

Figure 13.2: A Selection of Monitoring Systems

Of the systems mentioned in Figure 13.2, only PQL [133], Hawk/Eagle [53], and RuleR [20] provide logical formalisms with greater than finite-state power. Hawk/Eagle adopts a Turing-complete fix-point logic, but it has problems with large programs because it does not garbage collect the objects used in monitoring. In addition, Hawk/Eagle is not publicly available³. Because of this and the fact that Hawk/Eagle has not been run on DaCapo [23] with the same properties, we cannot compare JavaMOP with our new string rewriting systems plugin with Hawk/Eagle. RuleR is a rule-based monitoring system which has the ability to also specify Turing complete properties. The current implementation of RuleR is not built for efficiency, and is, additionally, not publicly available. PQL is not Turing-complete, and performance comparisons with PQL using an older, less efficient, version of JavaMOP can be found in [134]. String rewriting was used in the context of monitoring for detection of malware in [22]. This was, in many ways, the inspiration for adding string rewriting to MOP. However, the string rewriting patterns allowed in that work were regular (i.e., can capture only regular languages), while our goal is to provide a true Turing-complete logical

³[16] makes an argument for the inefficiency of Hawk/Eagle. Since Hawk/Eagle is not publicly available (only its rewrite based algorithm is public [53]), the authors of Hawk/Eagle kindly agreed to monitor some of the simple properties from [26]. We have confirmed the inefficiency claims of [16] with the authors of Hawk/Eagle.

formalism for parametric monitoring.

MOP [40, 39] is an extensible runtime verification framework that provides efficient, logic-independent support for parametric specifications. JavaMOP is an instance of MOP for the Java programming language. It allows the developer to specify desired properties using formal specification languages, along with code to execute when properties are matched or fail to match. Monitoring code is then automatically generated from the specified properties and integrated together with the user-provided code into the original system.

MOP is a highly extensible and configurable runtime verification framework. The user is allowed to extend the MOP framework with his/her own logics via *logic plug-ins* which encapsulate the monitor synthesis algorithms. This extensibility of MOP is supported by an especially designed layered architecture [39], which separates monitor generation and monitor integration. By standardizing the protocols between layers, modules can be added and reused easily and independently. MOP also provides efficient and logic-independent support for *parametric* parameters [38], which is useful for specifying properties related to groups of objects. This extension allows associating parameters with MOP specifications and generating efficient monitoring code from parametric specifications with monitor synthesis algorithms for non-parametric specifications. MOP's generic support for parametric patterns simplified our SRS plug-in's implementation.

The JavaMOP instance provides two interfaces: a web-based interface and a command-line interface, providing the developer with different means to manage and process JavaMOP specifications. AspectJ [114] is employed for monitor integration: JavaMOP translates outputs of logic plug-ins into AspectJ code, which is then merged within the original program by an AspectJ compiler. Seven logic-plug-ins are currently provided with JavaMOP: finite state machines, extended regular expressions, context-free grammars, past time linear temporal logic, linear temporal logic with past and future operators, past time linear temporal logic with calls and returns, and, now, string rewriting systems. Descriptions of the first six plugin-ins can be found in [135].

13.3 Monitoring SRS Specifications

In this section, we present some basic notation for string rewriting systems and our string rewriting algorithm which was implemented as a logic plugin in the MOP framework.

13.3.1 Preliminaries

We refer the reader to [27] for an in-depth presentation of string rewrite systems. For an alphabet Σ , a *string rewriting system* (SRS) is a binary relation, R , on Σ , that is, a subset of $\Sigma^* \times \Sigma^*$. The set $\{l \in \Sigma^* \mid (l, r) \in R\}$ is called the *domain* of R , denoted $dom(R)$, while similarly the set $\{r \in \Sigma^* \mid (l, r) \in R\}$ is called the *range*, denoted $range(R)$. We refer here to any element $(l, r) \in R$ as a *rule* in R , any $l \in dom(R)$ as a *left hand side (LHS)* of a rule in R , and any $r \in range(R)$ as a *right hand side (RHS)* of a rule in R . In our SRS specifications in this paper and in JavaMOP, rules $(l, r) \in R$ are written using the earlier shown syntax “ $l \rightarrow r$ ”.

The *single-step reduction relation* on Σ^* that is induced by R is defined as: for any $u, v \in \Sigma^*$, $u \rightarrow_R v$ if and only if there exists $(l, r) \in R$ such that for some $x, y \in \Sigma^*$, $u = x l y$ and $v = x r y$. The *reduction relation* on Σ^* induced by R is the reflexive, transitive closure of \rightarrow_R and is denoted by \rightarrow_R^* . If for $x, y \in \Sigma^*$, $x \rightarrow_R^* y$ and y is irreducible, y is a *normal form* for x . R is *confluent* if there is only one such y for any given x , regardless of the order in which rules are applied.

In our SRSs in MOP, the symbols $s \in \Sigma$ correspond to either *events* of our property or symbols that appear in the RHS of rules in R . We call our string rewriting systems *deterministic* because the same normal form will always be chosen in the presence of a non-confluent R . Specifically, rules are applied left-to-right, with the smallest rule matching first in the case of overlap (e.g., for LHSs $a a$ and $a a b$, the rule with $a a$ as its LHS will always be applied first, starving the other rule). In the case of a conflict that is not resolved by the above, the order of rules in the SRS specification is used to determine which rule to apply (e.g., if two rules have the same LHS, the one specified first will always be applied).

13.3.2 String Rewriting Algorithm Overview

There are two major parts to our SRS algorithm:

1. Finding matches of the LHSs of rules; and
2. Performing replacements with RHSs of rules.

To make replacements as efficiently as possible, the string of events/symbols that we rewrite is a linked list of the `SpliceList` class, which was specially created for our purposes to allow constant time replacement of a section of the list with another list (splicing). The `SpliceList` class has a special type of

iterator defined for it, called the `SLiterator`, that does *not* follow the normal iterator interface in Java.

Rather than only having `next()` and `hasNext()` methods, the `SLiterator` has `next(int i)`, which moves the `SLiterator` forward `i` times and returns true if it is successful (i.e., does not reach the end of the `SpliceList`), and `get()`, which returns the current element that the `SLiterator` points to. `SLiterator` also has a method, `splice(SLiterator second, SpliceList replacement)`, which takes another `SLiterator` to the same `SpliceList` and replaces the sequence denoted by those two `SLiterators`, inclusively, by a specified sequence replacement. It is because of the inclusive nature of the `splice` method that the `SLiterator` must have a method to retrieve its current element without advancing. The `splice` method makes it imperative for our string matching algorithm to maintain `SLiterators` to the beginning and end of the current LHS under consideration.

In Section 13.3.3 we discuss how this matching occurs using a modification of the Aho-Corasick string searching algorithm [4] that, unlike the base algorithm, keeps track of the beginning of a match, so that rewrites can be performed in constant time (after copying the RHS in time proportional to its length). To make the paper self-contained, we give all the necessary information regarding the Aho-Corasick algorithm, rather than only this modification, but the modification is clearly delineated. To our knowledge, this is the first time any variation on the Aho-Corasick algorithm has been used in string rewriting, and no implementations of SRSs exist, that we could find. In Section 13.3.4, we present an in-depth explanation of how the pattern matching fits into the string rewriting algorithm and how we optimize string rewriting to avoid considering sequences that cannot match any LHS.

13.3.3 Pattern Match Automata

The pattern match automata used by our string rewriting process, as mentioned, is a modification of the Aho-Corasick algorithm for finding strings in text [4]. The Aho-Corasick algorithm, which was originally not designed for string rewriting, is able to find all matches in a string in one linear pass, rather than performing separate passes for each rule LHS as would a naive matching algorithm. Our modification of the algorithm allows us to correctly adjust the `SLiterator` to the beginning of our current match, facilitating quick rewrites.

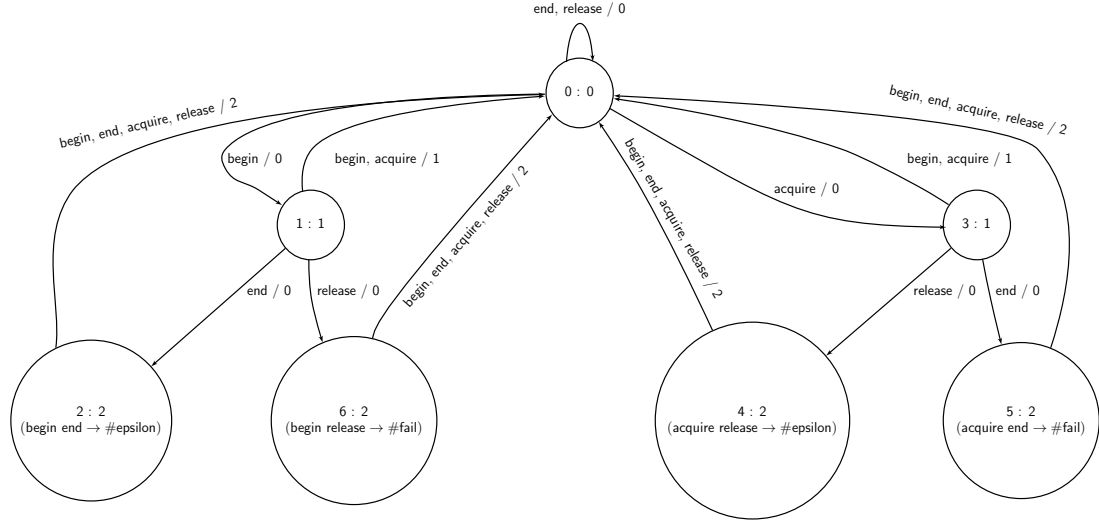


Figure 13.3: Pattern match automaton for the SAFELOCK property (see Section 13.1.1)

Using Pattern Match Automata

Figure 13.3 shows the pattern match automaton for the SAFELOCK property. Each node has at least its state number and state depth, listed as a pair *number:depth*. The depth is used in two places in the automata generation algorithm, and simply states how many symbols (events) have been processed since the start state in one of the LHSs of the rewrite rules in our SRS. This will be explained in more detail below. Additionally, states which correspond to matching the left hand side of a given rule also display that rule, e.g., in state 6, the `begin release → #fail` rule is matched. Each edge is marked by the list of symbols that cause that transition, as well as a number following a “/”. That number, which we refer to as the *action*, is the number of times to increment the *first* SLiterator except in the self-transitions of state 0. When a self-transition in state 0 occurs, the *first* iterator must be incremented once. When a forward transition with “/ 0” is encountered, a transition to the next state is made, and the next input is considered. If the transition is suffixed with something *other* than 0, the transition must be a backward transition, and the same symbol that is currently under consideration must be evaluated in the next state. This is why we handle self-transitions in state 0 as a special case, if it were suffixed with “/ 1” and handled as a

backward transition, the same symbol would be considered infinitely.

Figure 13.4 shows the pseudocode for pattern matching using a given pattern match automaton. The only global variable for the algorithm is the given `PatternMatchAutomaton`, *pma*. The algorithm begins by initializing the *first* and *second* `SLiterators` to the beginning of the argument `SpliceList l`, using the *head()* method. The local *currentState* is initialized to the initial machine state, here represented as 0⁴. The while loop beginning on line 10 will only exit when the end of *l* is reached, denoted by the `break` statements on lines 20 and 25. We know that the end of *l* is reached on lines 20 and 25 when the `next(int i)` method returns false. We never need to check if *first.next* returns false because it may never advance past *second* due to the construction of the `PatternMatchAutomaton`. Lines 17–22 cover the self transition to state 0 mentioned earlier, while lines 23–27 represent a normal forward transition. 23–27 are a forward transition because the *action* of the transition is 0. As mentioned earlier, the only difference between the 0 self-transition and a forward transition is that in the self-transition the *first* `SLiterator` need be incremented (line 18). Lines 28–30 handle a backward transition in the `PatternMatchAutomaton`. As expected, with a backward transition the *first* `SLiterator` is incremented a number of times specified by the *action* of *transition* and *second* is *not* incremented so that the same symbol will be considered in the next iteration of the loop. One interesting property of this algorithm is that if one pattern is a prefix of another, such as the patterns “*a a* \rightarrow *c*” and “*a a b* \rightarrow *d*”, both matches will be reported. This is undesirable behavior for rewriting because “*aa*” will be rewritten to *c* immediately and “*a a b*” should no longer be matchable. This will be accounted for in Section 13.3.4.

As an example of how the pattern match algorithm functions, suppose that the following series of events have been seen at a given point in a program: `begin begin acquire begin end`. At this point, the `SAFELOCK` property will experience its first match of a rule LHS. Figure 13.5 shows the state transitions as each symbol is considered, as well as the position of the *first* `SLiterator`. An important thing to note is that every time we transition back to state 0, the *first* `SLiterator` index is incremented by 1 (specified by the back transitions), and the symbol is evaluated again in state 0. In general, back transitions need not be to state 0, as we shall see. At the end of the input, the algorithm is in state 2, which matches the rule `begin end \rightarrow #epsilon`. The *first* `SLiterator` correctly points to index 3, which is the last `begin` event. The

⁴It is actually a class that may contain a matched rule, as we can see in Figure 13.3.

```

1  globals PatternMatchAutomaton pma
2  locals SListIterator first, second
3      State currentState, nextState
4      Symbol symbol
5      Transition transition
6  procedure match(SpliceList l)
7      first  $\leftarrow l.head()$ 
8      second  $\leftarrow l.head()$ 
9      currentState  $\leftarrow 0$ 
10 while (true){
11     if (currentState.hasMatch()){
12         //signal match
13     }
14     symbol  $\leftarrow second.get()$ 
15     transition  $\leftarrow pma.get(currentState, symbol)$ 
16     nextState  $\leftarrow transition.state$ 
17     if (nextState = 0){
18         first.next(1)
19         if ( $\neg second.next(1)$ ){
20             break
21         }
22     }
23     else if (transition.action = 0){
24         if ( $\neg second.next(1)$ ){
25             break
26         }
27     }
28     else {
29         first.next(transition.action)
30     }
31     currentState  $\leftarrow nextState$ 
32 }

```

Figure 13.4: Pattern Match Algorithm

current state	symbol	next state	<i>first</i> index
0	begin	1	0
1	begin	0	1
0	begin	1	1
1	acquire	0	2
0	acquire	3	2
3	begin	0	3
0	begin	1	3
1	end	2	3

Figure 13.5: A run of the pattern match algorithm on begin begin acquire begin end

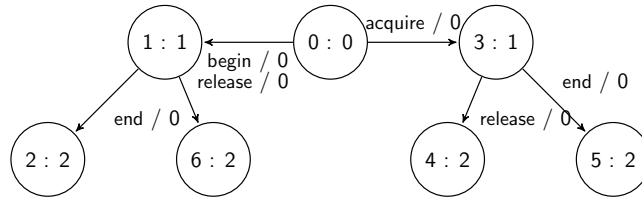


Figure 13.6: Forward Transitions for SAFELOCK (matched rules omitted)

second SLiterator always points at the current input, which is end. These SLiterators can then be used to quickly replace begin end with #epsilon, as we will see in Section 13.3.4.

Generating Pattern Match Automata

There are two main phases to the creation of pattern match automata. In the first phase the forward transitions of the automaton are created. In the second phase, all of the backward transitions and the self-transition that (almost) always exists in state 0 are added. During the computation of the backward transitions, the actions for the backward transition are also computed and added to the backward transitions. As mentioned, only backward transitions ever have non-0 actions, since they correspond to places in the automaton where there is a switch from matching one potential set of LHSs of rules to another. For instance, in Figure 13.5, between the third

begin and the first acquire, there is a switch from potentially matching $\{\text{begin end, begin release}\}$ to $\{\text{acquire end, acquire release}\}$, which requires no longer considering the begin event for match purposes, thus the action of 1.

To create the forward transitions for an automaton, we add one path that corresponds directly to the left hand side of each rule in our string rewriting system. We add these paths one at a time, and reuse as many states as possible. Each forward transition is assigned the action 0. Figure 13.6 shows the forward transitions for the pattern match automaton originally presented in Figure 13.3. For each LHS, we begin at state 0 and add a transition for the first symbol. Because all patterns SAFELock begin with either `begin` or `acquire`, we have only two transitions, one labeled with `begin` and one labeled with `acquire`. We continue to transitively add transitions based on the remainder of each LHS. For the two rule LHSs beginning with `begin`, one ends with `end` and the other ends with `release`, so there are two transitions out of state 1 labeled accordingly. As each new state is added to the machine during the forward transition phase, the depth of the state is recorded. The depth is simply the number of symbols from state 0. For instance, state 6 is at depth 2, since two symbols, `begin` followed by `end`, lead to state 6. The largest depth always corresponds to the longest rule LHS.

In the second phase, the self-transition on state 0 is added first, if needed. The self-transition is only necessary if there is not a forward transition out of state 0 for every symbol used in the SRS or specified by the JavaMOP front end⁵.

After potentially adding the self-transition in state 0, the backward transitions are added to the pattern match automaton. Backward transitions are only added from a given state for symbols that do not have forward transitions out of that state. All backward transitions from a given state, s , will go to the same place, so we define $fail(s) = s'$, where s' is the destination of a backward transition out of s . To find the destination for the backward transitions out of a state in pattern match automaton pma with depth d , we consider each state r of depth $d - 1$ and perform the following actions, transitions are added in depth first order [4]:

1. If $pma.get(r, a)$ is a backward transition for all symbols a , do nothing.
2. Otherwise, for each symbol a such that $pma.get(r, a) = s$, do the following:

⁵JavaMOP allows one to define events that do not appear in the specified property; these will correspond to symbols that are never rewritten by the specified SRS.

- (a) Let $s' = fail(r)$.
- (b) Compute $s' \leftarrow fail(s')$ until such point as $pma.get(s', a).action = 0$. Because state 0 must have either a forward transition or a self-transition for every symbol, such an s' must exist.
- (c) For all a' such that $pma.get(s, a')$ has no forward transition, assign $pma.get(s, a').state = s'$, **$pma.get(s, a').action = s.depth - s'.depth$** .

The procedure above is essentially the same as [4]. The part in bold is specific to our algorithm for string rewriting. The action is assigned as such because the depth of a given state represents the number of symbols processed since state 0 in the automaton, thus the difference in the depths tell us the number of symbols that we need to skip with the *first* SIterator in Figure 13.4. While the pattern match automaton for SAFELOCK has backward transitions that only go to state 0, as mentioned, this is not always the case in general. When the suffix of one LHS overlaps with the prefix of another, backward transitions that do not go back to state 0 are generated. An example of this can be seen in Figure 13.7, where the SRS in question is $b\ a\ a \rightarrow \#epsilon, a\ a\ c \rightarrow \#epsilon$. Because $b\ a\ a$ and $a\ a\ c$ have a suffix/prefix overlap, the backward transitions from state 3 at depth 3 go to state 5 at depth 2, resulting in an action of only 1. For example, consider input $b\ a\ a\ c$. When we switch from matching $b\ a\ a$ to matching $a\ a\ c$, which occurs between states 3 and 5, we wish to only “forget” the b at the beginning, an action of 1. Note that when we use the rule application order of “smallest left hand side”, this transition will never be taken because $b\ a\ a$ will be immediately rewritten. We include these transitions in the automaton for future rewrite orders.

13.3.4 Rewriting using Pattern Match Automata

The rewriting algorithm we use to monitor SRS’s is presented in Figure 13.8. Not pictured in Figure 13.8, is the action of the monitor itself. As any monitoring algorithm in the MOP framework, events arrive one at a time. As each event occurs, we add it—as a symbol representing that event—to a SpliceList that contains the results of rewriting previous sequences of events. Additionally, if any rules make use of the $\hat{\ }$ symbol, it will be added to the beginning of the SpliceList and treated as a normal symbol by the rewriting

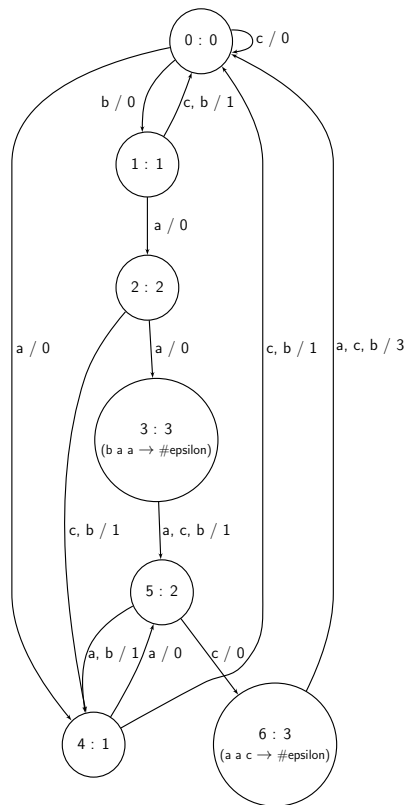


Figure 13.7: A pattern match automaton with overlap


```

1  globals PatternMatchAutomaton pma
2  locals SLLiterator first, second, last
3      State currentState, nextState
4      Symbol symbol
5      Transition transition
6      boolean changed pastLast
7  procedure match(SpliceList l)
8  do {
9      first  $\leftarrow$  l.head()
10     second  $\leftarrow$  l.head()
11     currentState  $\leftarrow$  0
12     changed  $\leftarrow$  false
13     pastLast  $\leftarrow$  false
14     while (true){
15         if (currentState.hasMatch()){
16             if (currentState.match = #succeed){
17                 // raise succeed
18             }
19             if (currentState.match = #fail){
20                 // raise fail
21             }
22             first.splice(second, currentState.match)
23             nextState  $\leftarrow$  0
24             changed  $\leftarrow$  true
25             pastLast  $\leftarrow$  false
26             last  $\leftarrow$  second
27             second  $\leftarrow$  first.copy()
28         }
29         symbol  $\leftarrow$  second.get()
30         transition  $\leftarrow$  pma.get(currentState, symbol)
31         nextState  $\leftarrow$  transition.state
32         if (nextState = 0){
33             first.next(1)
34             if ( $\neg$ second.next(1)){
35                 break
36             }
37         }
38         else if (transition.action = 0){
39             if ( $\neg$ second.next(1)){
40                 break
41             }
42         }
43         else {
44             first.next(transition.action)
45         }
46         if ( $\neg$ changed){
47             if (second = last){
48                 pastLast  $\leftarrow$  true
49             }
50             if (pastLast and nextState = 0){
51                 return
52             }
53         }
54         currentState  $\leftarrow$  nextState
55     }
56 } while (changed)

```

Figure 13.8: Rewriting Algorithm

algorithm. As for uses of \$, the current event must be added *before* \$⁶.

After an event is added to the `SpliceList`, the algorithm in Figure 13.8 is evaluated to completion before another event can be accepted. The algorithm is similar to the pattern match procedure of Figure 13.4. The changes are in bold. There are three main changes: the inclusion of a loop that ensures that a normal form is reached, the actual rewriting step itself, and a section that recognizes early termination.

The first new control structure to notice is the `do...while` loop from line 8 to 56. This loop ensures that rewriting continues until there is a pass through the loop in which nothing changes, i.e., the string is in normal form. The new boolean variable, *changed*, controls this loop. It is set to false at the beginning of an iteration of the `do...while` loop, and to true on line 24, which is only executed when a rewrite occurs.

Lines 15–28 perform the actual rewriting step. The element *match* of a `State` contains the right hand side of the rule matched in that `State`. If the *match* is one of the two special keywords `#succeed` or `#fail`, a success or fail handler is executed, as appropriate, and rewriting terminates. If either handler is executed, the monitor is considered dead unless it is reset (see [135]). If *match* is something else, the `splice` method is called on line 22. The `splice` method is a special method of `SLiterator` that replaces a range specified by the `this` and an argument `SLiterator` with the argument sequence. Here the range is specified by *first* and *second*, and *currentState.match* is passed as the replacement. Note that if the right hand side of the rule is `#epsilon`, it is represented as an empty sequence, which `splice` is able to handle. The `splice` method also correctly sets *first* and *second* to point to the beginning and end of the spliced in *match* sequence, or the next symbol if *match* was `#epsilon`. On line 26, we set *last* to *second*, so that *last* points to the end of the last replacement, this will be used to determine early termination. Then, on line 27, *second* is set as a copy of *first*. This ensures that segments of string which are transitively rewritten will be rewritten immediately. Because `splice` changes the `SpliceList`, it is important to set *currentState* back to state 0 because any matching will occur in the newly rewritten segment of the `SpliceList`.

In the last new addition to the match algorithm, from lines 46 to 53, we test for early termination of the algorithm. The idea here is to exit early if we enter a segment of the `SpliceList` that we know for certain cannot be

⁶Because of this there is a very small performance hit for using \$ in a rule, but ^ is essentially free.

event	initial l	l in normal form
begin	begin	begin
end	begin end	#epsilon
begin	begin	begin
acquire	begin acquire	begin acquire
release	begin acquire release	begin
acquire	begin acquire	begin acquire
end	begin acquire end	#fail

Figure 13.9: An SRS monitoring run for SAFELock

rewritten. This happens when we reach a point that is past the end of the *last* `SLIterator`, which was set in a previous iteration, no rewrites have occurred in the current iteration, and *currentState* returns to 0. The first two requirements are fairly straight-forward: if a change occurs, new matches are possible, and if we are in a segment of the `SpliceList` before the last rewrite, we are still investigating symbols that are potentially new. However, if there is no rewrite in the current iteration and we are past the last change from the previous iteration, we are seeing symbols that were seen in the previous iteration with no change. The last condition, that we must return to State 0, is more subtle. The reason for this is that there could have been a rewrite in the last iteration that inserted a segment that appears in the middle of a left hand side of one of the rules. A simpler way to look at this requirement is that if *pma* is not in state 0 it is actively matching *something*. This condition for early termination can lead to an unbounded amount of saving, as the `SpliceList` can be of an unbounded length.

Figure 13.9 shows a monitor run as non-parametric events for SAFELock arrive. The non-parametric events are dispatched to the correct monitor instance by the indexing of JavaMOP (or whatever projection method is used in future language instances of MOP). The first column shows the arriving event, the second column shows the state of the `SpliceList` l before any rewriting, and the last column shows the normal form for l after the rewriting algorithm of Figure 13.8 has run. After the last event a failure has occurred, and the fail handler will execute.

Benchmark	Orig. (ms)	HasNext		SafeSyncCol		SafeSyncMap		UnsafeIter		UnsafeMapIter	
		ERE	SRS	ERE	SRS	ERE	SRS	ERE	SRS	ERE	SRS
avro	2317*	194	227	35*	103*	28	120*	253*	288*	41	134*
batik	773	0	6	5	11	9	-1	5	3	1	2
eclipse	11749	-1	-2	-2	-4	-1	-3	-2	-2	-2	-2
fop	251	922	2091	26	24	21	20	34	57	28	42
h2	3860	9	15	6	2	0	4	15	22	8	24
jython	1400	3	4	3	3	4	2	16	18	3	3
luindex	478	2	-1	2	0	0	4	1	2	0	-5
lusearch	581	1	3	-1	1	3	3	46	46	2	0
pmd	1441	27	117	139	137	10	17	72	148	177	199
sunflow	1222	5	8	0	-1	6	-3	-4	4	0	3
tomcat	1068	2	4	3	3	3	1	2	2	2	2
tradebeans	4618	2	1	-1	-3	-1	2	4	-2	-2	-1
tradesoap	3213	1	-1	1	-1	0	-2	1	0	0	0
xalan	359	5	1	5	1	6	3	90	172	7	8

Figure 13.10: Comparison of JavaMOP with extended regular expressions (ERE) and with the same properties expressed as string rewriting systems (SRS): average *percent* overhead (convergence within 3% except those marked with *)

13.4 Evaluation

An important thing to note about SRS execution is that it may add an unbounded amount overhead to a program execution in full generality, since it is Turing-complete. Because of this, our evaluation focuses on two specific types of experiments: first we show how it compares, within the context of JavaMOP, to finite-state logics on the DaCapo benchmark suite [23]. Then we give a comparison of our underlying SRS rewrite engine against the Maude [44] term rewriting engine, modulo associativity. The goal of the first evaluation is to show that SRS monitoring is efficient enough to be used in large programs, being not much less efficient than finite-state logics⁷ when monitoring finite-state properties. However, it should be stressed that a fully recursively enumerable safety property may add an unbounded amount of overhead, or even not terminate. The goal of the second experiment is to show that our SRS implementation is more efficient than the state-of-the-art⁸.

All experiments were performed on a machine with a 3.82GHz Intel® Core™ i7 970 hexcore with Hyper-Threading (12 hardware threads) and 24 GB of ram. Ubuntu 11.10 64 bit was used as the operating system and version 9.12 of DaCapo was used as the benchmark suite, with default inputs and the -converge option to gain convergence within 3%. OpenJDK version 1.6.0.23 as the Java virtual machine. All compiled JavaMOP specs were weaved into DaCapo using ajc 1.6.11. Maude 2.6 was used for comparison with Maude.

The following properties were used in the DaCapo experiments. The SRS versions of them (shown below) are new, while the extended regular expression versions were borrowed from [26, 25, 38, 134].

- HASNEXT: Do not use the next element in an Iterator without checking for the existence of it (see Section 13.1.1);
- SAFESYNCCOL: If a Collection is synchronized, then its iterator also should be accessed synchronously:

<code>sync asyncCreateliter</code>	<code>→</code>	<code>#fail</code>
<code>sync syncCreateliter accesslter</code>	<code>→</code>	<code>#fail</code>

⁷In this case, extended regular expressions.

⁸Note that Maude is more general than our SRS engine, but there is a price for that generality, and general term rewriting makes little sense in the context of MOP event traces.

N	Maude Time (ms)	SRS Time (ms)
100	42	33
1000	37038	236
5000	DNF	7112
10000	DNF	26132

Figure 13.11: Comparison of maude versus SRS rewrite. DNF: did not finish in one hour

- SAFESYNCMAP: If a Collection is synchronized, then its iterators on values and keys also should be accessed in a synchronized manner:

sync createSet asyncCreateltr → #fail
sync createSet syncCreateltr accessltr → #fail

- UNSAFEITER: Do not update a Collection when using the Iterator interface to iterate its elements:

update use → #fail
use use → use
update update → update
createltrator → #epsilon

- UNSAFEMAPITER: Do not update a Map when using the Iterator interface to iterate its values or its keys:

update use → #fail
use use → use
update update → update
createltrator → #epsilon
createCollection → #epsilon

For the comparison with Maude, strings of equal numbers of 2's, 1's, and 0's, with the 2's preceding the 1's preceding the 0's were generated, and the following rewrite system applied. Note, that the language of strings that reduce to #epsilon with this rewrite system is strictly context-sensitive.

1 0 → 0 1 2 0 → 0 2
2 1 → 1 2 0 1 → 3
1 3 → 3 1 3 0 → 0 3
3 2 → #epsilon 2 3 → #epsilon

Figure 13.10 shows a comparison of finite-state properties specified in JavaMOP using ERE and SRS. The first column shows the individual DaCapo [23] benchmarks, and the second column shows runtime of the original uninstrumented benchmarks in milliseconds. All other columns are *percent* overhead. Each benchmark-property pair converged to within 3% except the instances of *avroa* marked with *. The results presented for *avroa* that did not converge are the average of twenty runs with outliers removed, but they are still not as trustworthy as the converging results. This lack of convergence is a problem on highly multi-threaded machines. We can see that even the uninstrumented, original run, fails to converge. Negative overheads are the result of noise in the experimental settings and changes in code layout due to instrumentation resulting in slightly more efficient programs.

Overall, the average overhead on the DaCapo benchmark suite was 58% for SRS, while it was 33% for ERE. When *fop-HASNEXT*—which has, by far, the worst overhead of any trial—is removed from both, the overhead drops to 29% and 20%, respectively. It must be noted, that the properties we use are specifically selected for generating large overheads; they are very intensive properties that generate *many events* (see [111]). The overhead numbers are slightly larger than reported in previous papers because we have moved to a multi-threaded, and quite simply faster, machine. The monitors in JavaMOP must be synchronized, which results in higher overhead for programs that actually make use of multiple threads. Any monitoring system must do the same thing if the monitors are for cross-thread properties (like all of those properties used here). In most of the benchmark/property pairs, the performance of ERE and SRS are very comparable. For *pmd-HASNEXT* and *avroa-SAFESYNCMAP*, SRS shows more than three times the overhead of ERE, but for all other trials SRS is never more than three times worse.

Figure 13.11 shows the comparison of Maude to our SRS engine with the rewrite system discussed above. N refers to the number of each digit, i.e., N=100 has 300 characters in it: 100 each of 2, 1, and 0. As we can see from the results, our SRS engine runs in 78% of the time of maude at N=100. At N=1000, our SRS engine runs in .006% of the time of Maude. With larger inputs, Maude fails to complete in an hour, while our SRS engine takes less than 30 seconds on every tested input.

Acknowledgments

The work presented in this paper was supported in part by the NSF grant CCF-1218605, the NSA grant H98230-10-C-0294, the DARPA HACMS program as SRI subcontract 19-000222, the Rockwell Collins contract 4504813093, and the (Romanian) SMIS-CSNR 602-12516 contract no. 161/15.06.2010.

13.5 Conclusion

We provided the first means to efficiently monitor parametric Turing-complete specifications using string rewriting systems. By using a modified version of the Aho-Corasick string matching algorithm and a means to terminate the rewriting process early, the resultant string rewriting algorithm is quite practical, as shown in our extensive evaluation⁹.

The average overhead on the DaCapo benchmark suite was 58% for finite state properties monitored using SRS, while it was 33% using ERE plugin to monitor the same properties. When the largest benchmark/property pair is removed from both, the overhead drops to 29% and 20%, respectively. This shows that the SRS plugin can be efficient when monitoring reasonable properties on large programs. We must stress however that arbitrarily complex properties may add arbitrary overhead, regardless of the efficiency of the monitoring algorithm or implementation there of.

A less extensive comparison of our core string rewriting algorithm with the term rewrite engine Maude, which provides implicit support for string rewriting through its rewriting modulo associativity, suggests that our approach vastly outperforms the state-of-the-art in rewriting, when restricted to string rewriting¹⁰.

⁹Special thanks to Dongyun Jin for help with DaCapo experimental settings.

¹⁰The full generality of term rewriting provided by Maude makes little sense when applied to monitoring safety properties, which necessarily operate on strings (traces) of events.

Chapter 14

Parametric Property Monitoring

Chapter 15

Predictive Runtime Analysis

Chapter 16

Static Analysis to Improve Runtime Verification

Chapter 17

Semantics-Based Runtime Verification

17.1 Defining a Formal Semantics

17.2 Semantics-Based Symbolic Execution

17.3 Program Verification as Exhaustive Runtime Verification

Chapter 18

Conclusion and Future Work

18.1 Safety Properties and Monitoring

Chapters 3 and 4 presented a comprehensive study of safety properties and of their monitoring, using a uniform formalism and notation. Technically, there were two novel contributions. First, it introduced the notion of a *persistent* safety property, which is the finite-trace correspondent of an infinite-trace safety property, and used it to show the cardinal equivalence of the various notions of safety property encountered in the literature. Second, it rigorously defined the problem of monitoring a safety property, and it showed that it can be arbitrarily hard. These results established a firm foundation for studying safety properties and corresponding monitors and algorithms for various domains of interest, where requirements can be expressed using domain-specific formalisms, such as future-time and past-time temporal logics, context-free grammars, push-down automata, and so on.

Bibliography

- [1] MOP website, LTL plugin. <http://fsl.cs.uiuc.edu/mop/logic-plugins/ltl>.
- [2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *LICS*, pages 165–175. IEEE Computer Society, 1988.
- [3] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
- [4] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.
- [5] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In Richard P. Gabriel, editor, *ACM Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 345–364. ACM Press, 2005.
- [6] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05*, 2005.
- [7] Bowen Alpern and Fred B. Schneider. Defining liveness. *IPL*, 21(4):181–185, 1985.
- [8] Rajeev Alur, Kousha Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 467–481. Springer, 2004.
- [9] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

- [10] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. *Theoretical Computer Science*, 143(1):51–72, 1995.
- [11] Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. In *POPL*, pages 189–198, 1987.
- [12] C. Artho, A. Biere, and K. Havelund. High-level data races. In *The 1st International Workshop on Verification and Validation of Enterprise Information Systems (VVEIS'03)*, April 2003.
- [13] Cyrille Artho, Doron Drusinsky, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Roşu, and Willem Visser. Experiments with Test Case Generation and Runtime Analysis. In *Proc. of ASM'03: Abstract State Machines*, volume 2589 of *Lecture Notes in Computer Science*, pages 87–107, Taormina, Italy, March 2003. Springer.
- [14] Cyrille Artho, Doron Drusinsky, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Păsăreanu, Grigore Roşu, Willem Visser, and Rich Washington. Automated Testing using Symbolic Execution and Temporal Monitoring. *Theoretical Computer Sci.*, to appear, 2005.
- [15] Formal System Laboratory at UIUC. ptCaRet MOP Logic Plugin. <http://fsl.cs.uiuc.edu/index.php/Special:JavaMOPPTCARETOnline>.
- [16] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making Trace Monitors Feasible. In *OOPSLA'07*, 2007.
- [17] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. of TACAS'01: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, Genova, Italy, April 2001.
- [18] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of VMCAI'04*, volume 2937 of *LNCS*, pages 44–57, 2004.
- [19] Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors. *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*. Springer, 2010.
- [20] Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from eagle to ruler. *J. Logic Computation*, pages exn076+, November 2008.

- [21] David A. Basin, Matús Harvan, Felix Klaedtke, and Eugen Zalinescu. Monopoly: Monitoring usage-control policies. In *Runtime Verification (RV'11)*, pages 360–364, 2011.
- [22] Philippe Beaucamps, Isabelle Gnaedig, and Jean-Yves Marion. Behavior abstraction in malware analysis. In *Runtime Verification (RV'10)*, pages 168–182, 2010.
- [23] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06*, pages 169–190. ACM, 2006.
- [24] Eric Bodden. J-LO, a tool for runtime-checking temporal assertions. Master’s thesis, RWTH Aachen University, 2005.
- [25] Eric Bodden, Feng Chen, and Grigore Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09)*, pages 3–14. ACM, 2009.
- [26] Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP'07*, volume 4609 of *LNCS*, pages 525–549, 2007.
- [27] Ronald V. Book and Friedrich Otto. *String-rewriting systems*. Texts and monographs in computer science. Springer, 1993.
- [28] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and Proof in Membership Equational Logic. *Theoretical Computer Science*, 236:35–132, 2000.
- [29] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [30] J R Büchi. *On a Decision Method in Restricted Second Order Arithmetic*. Logic, Methodology and Philosophy of Sciences. Stanford University Press, 1962.
- [31] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.
- [32] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

- [33] Swarat Chaudhuri and Rajeev Alur. Instrumenting C programs with nested word monitors. In *14th Workshop on Model Checking Software (SPIN)*, volume 4595 of *Lecture Notes in Computer Science*, pages 279–283. Springer, 2007. Tool paper.
- [34] Swarat Chaudhuri and Rajeev Alur. Instrumenting C programs with nested word monitors. In *Model Checking Software (SPIN'07)*, volume 4595 of *LNCS*, pages 279–283, 2007.
- [35] F. Chen and G. Roşu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. In *Proc. of RV'03: the Third International Workshop on Runtime Verification*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 106–125, Boulder, Colorado, USA, 2003. Elsevier Science.
- [36] Feng Chen, Marcelo d'Amorim, and Grigore Roşu. A Formal Monitoring-Based Framework for Software Development and Analysis. In *Proceedings of ICFEM'04*, volume 3308 of *LNCS*, pages 357–372, 2004.
- [37] Feng Chen, Marcelo D'Amorim, and Grigore Roşu. Checking and correcting behaviors of Java programs at runtime with Java-MOP. In *RV'05*, volume 144(4) of *ENTCS*, 2005.
- [38] Feng Chen, Patrick Meredith, Dongyun Jin, and Grigore Rosu. Efficient formalism-independent monitoring of parametric properties. In *Automated Software Engineering (ASE'09)*, pages 383–394. IEEE, 2009.
- [39] Feng Chen and Grigore Roşu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specif. and Implementation. In *RV'03*, volume 89(2) of *ENTCS*, 2003.
- [40] Feng Chen and Grigore Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *OOPSLA '07*, 2007.
- [41] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
- [42] M. Clavel. The ITP Tool. In *Logic, Language and Information. Proc. of the First Workshop on Logic and Language*, pages 55–62. Kronos, 2001.
- [43] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Towards Maude 2.0. In *3rd International Workshop on Rewriting Logic and its Applications (WRLA'00)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.
- [44] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

- [45] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
- [46] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. Maude Manual. <http://maude.cs.uiuc.edu>.
- [47] Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. The Maude System. In *Proc. of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 240–243, Trento, Italy, July 1999. Springer-Verlag. System Description.
- [48] Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. A Maude Tutorial, March 2000. Manuscript at <http://maude.csl.sri.com/papers>.
- [49] Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285:187–243, 2002.
- [50] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *Software Engineering and Formal Methods (SEFM’09)*, pages 33–37, 2009.
- [51] James Corbett, Matthew B. Dwyer, John Hatcliff, Corina S. Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proc. of ICSE’00: International Conference on Software Engineering*, Limerich, Ireland, June 2000. ACM Press.
- [52] M. Dahm. BCEL. <http://jakarta.apache.org/bcel>.
- [53] Marcelo d’Amorim and Klaus Havelund. Event-based runtime verification of Java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.
- [54] Marcelo d’Amorim and Grigore Roşu. Efficient monitoring of ω -languages. In *Proceedings of 17th International Conference on Computer-aided Verification (CAV’05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 364 – 378. Springer, 2005.

- [55] Marcelo d'Amorim and Grigore Roşu. Efficient monitoring of ω -languages. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *LNCS*, pages 364–378, 2005.
- [56] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A Deadlock Detection Tool for Concurrent Java Programs. *Software Practice and Experience*, 29(7):577–603, July 1999.
- [57] D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.
- [58] Doron Drusinsky. Temporal Rover. <http://www.time-rover.com>.
- [59] Doron Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *Proc. of CAV'03: Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 114–118, Boulder, Colorado, USA, 2003. Springer-Verlag.
- [60] Kousha Etessami and Gerard Holzmann. Optimizing Büchi Automata. In *Proc. of Int. Conf. on Concurrency Theory*, volume 1877 of *LNCS*, pages 153–167. Springer, 2000.
- [61] James Ezick. An Optimizing Compiler for Batches of Temporal Logic Formulas. In *Proceedings of ISSTA'04*, pages 183–194. ACM Press, 2004.
- [62] C. J. Fidge. Partial Orders for Parallel Debugging. In *Proc. of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed Debugging*, pages 183–194. ACM, 1988.
- [63] B. Finkbeiner and H. Sipma. Checking Finite Traces using Alternating Automata. In *Proc. of RV'01: The First International Workshop on Runtime Verification*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*, Paris, France, 2001. Elsevier Science.
- [64] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny Sipma. Collecting Statistics over Runtime Executions. In *Proc. of RV'02: The Second International Workshop on Runtime Verification*, volume 70 of *Electronic Notes in Theoretical Computer Science*, Paris, France, 2002. Elsevier.
- [65] Dov M. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In *Proceedings of the 1st Conference on Temporal Logic in Specification*, volume 398 of *LNCS*, pages 409–448. Springer, 1989.
- [66] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(1):1203–1233, September 2000.

- [67] Michael Garey. Optimal Binary Identification Procedures. *SIAM Journal on Applied Mathematics*, 23(2):173–186, 1972.
- [68] P. Gastin and D. Oddoux. LTL with Past and Two-Way Very-Weak Alternating Automata. In *Proceedings of MFCS'03*, number 2747 in LNCS, pages 439–448. Springer, 2003.
- [69] Marc Geilen. On the Construction of Monitors for Temporal Logic Properties. In *Proceedings of RV'01*, volume 55 of *ENTCS*. Elsevier Science, 2001.
- [70] Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Proc. of the 15th Workshop on Protocol Specification, Testing, and Verification*. North-Holland, 1995.
- [71] D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
- [72] Patrice Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proc. of POPL'97: the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, France, January 1997.
- [73] J. Goguen, K. Lin, and G. Roşu. Circular coinductive rewriting. In *Proceedings, Automated Software Engineering '00*, pages 123–131. IEEE, 2000. (Grenoble, France).
- [74] J. Goguen, K. Lin, and G. Rosu. Conditional circular coinductive rewriting with case analysis. In *Recent Trends in Algebraic Development Techniques (WADT'02)*, Lecture Notes in Computer Science, to appear, Frauenchiemsee, Germany, September 2002. Springer-Verlag.
- [75] Joseph Goguen, Kai Lin, Grigore Roşu, Akira Mori, and Bogdan Warinschi. An overview of the Tatami project. In *Cafe: An Industrial-Strength Algebraic Formal Method*, pages 61–78. Elsevier, 2000.
- [76] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Initial Algebra Semantics and Continuous Algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, January 1977.
- [77] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000.
- [78] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA'05*, pages 385–402. ACM, 2005.

- [79] Elsa Gunter and Doron Peled. Tracing the Executions of Concurrent Programs. In *Proc. of RV'02: Second International Workshop on Runtime Verification*, volume 70 of *Electronic Notes in Theoretical Computer Science*, Copenhagen, Denmark, 2002. Elsevier.
- [80] Elsa L. Gunter, Robert P. Kurshan, and Doron Peled. PET: An Interactive Software Testing Tool. In *Proc. of CAV'00: Computer Aided Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 552–556, Chicago, Illinois, USA, 2003. Springer-Verlag.
- [81] Elsa L. Gunter and Doron Peled. Using Functional Languages in Formal Methods: The PET System. In *Parallel and Distributed Processing Techniques and Applications*, pages 2981–2986. CSREA, 2000.
- [82] Kevin W. Hamlen, J. Gregory Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.
- [83] K. Havelund and G. Roşu. Java PathExplorer – A Runtime Verification Tool. In *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, Montreal, Canada, June 18 – 21, 2001.
- [84] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
- [85] K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
- [86] K. Havelund and G. Roşu. *Runtime Verification 2002*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2002. Proceedings of a *Computer Aided Verification (CAV'02)* satellite workshop.
- [87] K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Software Tools and Technology Transfer*, 6(2):158–173, 2004. (also TACAS'02, LNCS 2280).
- [88] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

- [89] Klaus Havelund, Scott Johnson, and Grigore Roşu. Specification and Error Pattern Based Program Monitoring. In *Proc. of the European Space Agency workshop on On-Board Autonomy*, Noordwijk, The Netherlands, October 2001.
- [90] Klaus Havelund, Scott Johnson, and Grigore Roşu. Specification and Error Pattern Based Program Monitoring. In *European Space Agency Workshop on On-Board Autonomy*, Noordwijk, The Netherlands, 2001.
- [91] Klaus Havelund, Michael R. Lowry, and John Penix. Formal Analysis of a Space Craft Controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, August 2001. An earlier version occurred in the Proc. of SPIN’98: the fourth SPIN workshop, Paris, France, 1998.
- [92] Klaus Havelund and Thomas Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000. Special issue containing selected submissions to SPIN’98: the fourth SPIN workshop, Paris, France, 1998.
- [93] Klaus Havelund and Grigore Roşu. Java PathExplorer – A Runtime Verification Tool. In *Proc. of i-SAIRAS’01: the 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, Montreal, Canada, June 2001.
- [94] Klaus Havelund and Grigore Roşu. Monitoring Java Programs with Java PathExplorer. In *Proc. of RV’01: the First International Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 97–114, Paris, France, July 2001. Elsevier Science.
- [95] Klaus Havelund and Grigore Roşu. *Workshops on Runtime Verification (RV’01, RV’02, RV’04)*, volume 55, 70(4), to appear of *ENTCS*. Elsevier, 2001, 2002, 2004.
- [96] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’02)*, volume 2280 of *LNCS*, pages 342–356. Springer, April 2002.
- [97] Klaus Havelund and Grigore Rosu. Efficient monitoring of safety properties. *STTT*, 6(2):158–173, 2004.
- [98] Klaus Havelund and Natarajan Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In *Proc. of FME’96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681, Oxford, England, 1996. Springer.

- [99] S. Hirst. A new algorithm solving membership of extended regular expressions. Technical report, The University of Sydney, 1989.
- [100] Gerard Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
- [101] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
- [102] Gerard J. Holzmann and Margaret H. Smith. A Practical Method for Verifying Event-Driven Software. In *Proc. of ICSE’99: International Conference on Software Engineering*, Los Angeles, California, USA, May 1999. IEEE/ACM.
- [103] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.
- [104] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [105] Jieh Hsiang. *Refutational Theorem Proving using Term Rewriting Systems*. PhD thesis, University of Illinois at Champaign-Urbana, 1981.
- [106] Jieh Hsiang. Refutational Theorem Proving using Term Rewriting Systems. *Artificial Intelligence*, 25:255–300, 1985.
- [107] Laurent Hyafil and Ronald L. Rivest. Computing Optimal Binary Decision Trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.
- [108] L. Ilie, B. Shan, and S. Yu. Fast algorithms for extended regular expression matching and searching. In *Proceedings of STACS’03*, volume 2607 of *LNCS*, pages 179–190, 2003.
- [109] L. Ilie, B. Shan, and S. Yu. Fast algorithms for extended regular expression matching and searching. In H. Alt and M. Habib, editors, *Proceedings of the 20th International Symposium on Theoretical Aspects of Computer (STACS 03)*, volume 2607 of *Lecture Notes in Computer Science*, page 179. Springer-Verlag, Berlin, 2003.
- [110] JavaCC. Url. http://www.webgain.com/products/java_cc.
- [111] Dongyun Jin, Patrick O’Neil Meredith, Dennis Griffith, and Grigore Roşu. Garbage collection for monitoring parametric properties. In *Programming Language Design and Implementation (PLDI’11)*, pages 415–424. ACM, 2011.
- [112] JTrek. Web page. <http://www.compaq.com/java/download>.

- [113] Yonit Kesten, Zohar Manna, Hugh McGuire, and Amir Pnueli. A Decision Algorithm for Full Propositional Temporal Logic. In *Proceedings of CAV'93*, volume 697 of *LNCS*, pages 97–109. Springer, 1993.
- [114] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP'01*, volume 2072 of *LNCS*, pages 327–353, 2001.
- [115] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
- [116] MoonZoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.
- [117] J.R. Knight and E.W. Myers. Super-pattern matching. *Algorithmica*, 13(1/2):211–243, 1995.
- [118] David Kortenkamp, Tod Milam, Reid Simmons, and Joaquin Fernandez. Collecting and Analyzing Data from Distributed Control Programs. In *Proc. of RV'01: First International Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*, Paris, France, 2001. Elsevier Science.
- [119] O. Kupferman and M. Y. Vardi. Freedom, Weakness, and Determinism: From Linear-Time to Branching-Time. In *Proc. of the IEEE Symposium on Logic in Computer Science*, pages 81–92, 1998.
- [120] O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In *Proc. of CAV'99: Conference on Computer-Aided Verification*, Trento, Italy, 1999.
- [121] O. Kupferman and S. Zuhovitzky. An Improved Algorithm for the Membership Problem for Extended Regular Expressions. In *Proc. of the International Symposium on Mathematical Foundations of Computer Science*, volume 2420 of *Lecture Notes in Computer Science*, 2002.
- [122] O. Kupferman and S. Zuhovitzky. An improved algorithm for the membership problem for extended regular expressions. In *Proc. of MFCS'02*, volume 2420 of *LNCS*, pages 446–458, 2002.
- [123] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
- [124] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, October 2001.

- [125] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.
- [126] Leslie Lamport. Logical foundation. In M. W. Alford, J. P. Ansart, G. Hommel, L. Lamport, B. Liskov, G. P. Mullery, F. B. Schneider, M. Paul, and H. J. Siebert, editors, *Distributed systems: Methods and tools for specification. An advanced course*, volume 190 of *LNCS*, pages 119–130. Springer-Verlag, 1985.
- [127] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
- [128] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.
- [129] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.
- [130] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, New York, 1995.
- [131] Nicolas Markey. Temporal Logic with Past is Exponentially more Succinct. *EATCS Bulletin*, 79:122–128, 2003.
- [132] Nicolas Markey and Philippe Schnoebelen. Model Checking a Path (Preliminary Report). In *Proc. of CONCUR’03: International Conference on Concurrency Theory*, volume 2761 of *Lecture Notes in Computer Science*, pages 251–265, Marseille, France, August 2003. Springer.
- [133] Michael Martin, V. Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA’07*, pages 365–383. ACM, 2005.
- [134] Patrick Meredith, Dongyun Jin, Feng Chen, and Grigore Roşu. Efficient monitoring of parametric context-free patterns. *Journal of Automated Software Engineering*, 17(2):149–180, June 2010.
- [135] Patrick O’Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, 2011. <http://dx.doi.org/10.1007/s10009-011-0198-6>.
- [136] José Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, pages 73–155, 1992.

- [137] José Meseguer. Membership Algebra as a Logical Framework for Equational Specification. In *Proc. of WADT'97: Workshop on Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61, Tarquinia, Italy, June 1998. Springer.
- [138] Bernard Moret. Decision Trees and Diagrams. *ACM Comp. Surv.*, 14(4):593–623, 1982.
- [139] G. Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(4):430–448, 1992.
- [140] Dennis Oddoux and Paul Gastin. LTL2BA. <http://www.liafa.jussieu.fr/~oddoux/ltl2ba/>.
- [141] T. O'Malley, D. Richardson, and L. Dillon. Efficient Specification-Based Oracles for Critical Systems. In *In Proceedings of the California Software Symposium*, 1996.
- [142] David Y.W. Park, Ulrich Stern, and David L. Dill. Java Model Checking. In *Proc. of the First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, June 2000.
- [143] Amir Pnueli. The Temporal Logic of Programs. In *Proc. of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
- [144] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-Based Test Oracles for Reactive Systems. In *Proceedings of the Fourteenth International Conference on Software Engineering, Melbourne, Australia*, pages 105–118, 1992.
- [145] G. Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.
- [146] G. Roşu and K. Havelund. Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae. Technical Report TR 01-08, NASA - RIACS, May 2001.
- [147] G. Roşu and M. Viswanathan. Testing extended regular language membership incrementally by rewriting. In *RTA '03*, volume 2706 of *LNCS*. Springer, 2003.
- [148] Grigore Roşu. On Safety Properties and Their Monitoring. Technical Report UIUCDCS-R-2007-2850, Dept. of Comp. Sci., Univ. of Illinois at Urbana-Champaign, 2007.
- [149] Grigore Roşu and Klaus Havelund. Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae. RIACS Technical report TR 01-08, January 2001.

- [150] Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.
- [151] Grigore Roşu and Klaus Havelund. Rewriting-Based Techniques for Runtime Verification. *Journal of Automated Software Engineering*, 2005. to appear.
- [152] Grigore Roşu. K: a rewrite-based framework for modular lang. design, semantics, analysis and implementation (V2). Technical Report UIUCDCS-R-2006-2802, 2006.
- [153] Grigore Rosu. An effective algorithm for the membership problem for extended regular expressions. In *Proceedings of the 10th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'07)*, volume 4423 of *LNCS*, pages 332–345. Springer-Verlag, 2007.
- [154] Grigore Rosu. On safety properties and their monitoring. *Sci. Ann. Comp. Sci.*, 22(2):327–365, 2012.
- [155] Grigore Roşu, Feng Chen, and Thomas Ball. Synthesizing monitors for safety properties – this time with calls and returns –. In *Workshop on Runtime Verification (RV'08)*, volume 5289 of *Lecture Notes in Computer Science*, pages 51–68. Springer, 2008.
- [156] J. J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR 98)*, volume 1466 of *Lecture Notes in Computer Science*, pages 194–218. Springer-Verlag, 1998.
- [157] Stefan Savage, Michael Burrows, Greg Nelson, Patrik Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
- [158] Fred B. Schneider. *On Concurrent Programming*. Springer, 1997.
- [159] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
- [160] Alper Sen and Vijay K. Garg. Partial Order Trace Analyzer (POTA) for Distributed Programs. In *Proc. of RV'03: the Third International Workshop on Runtime Verification*, volume 89 of *Electronic Notes in Theoretical Computer Science*, Boulder, Colorado, USA, 2003. Elsevier Science.
- [161] K. Sen and G. Roşu. Generating Optimal Monitors for Extended Regular Expressions. In *Proc. of RV'03: the Third International Workshop on Runtime Verification*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 162–181, Boulder, Colorado, USA, 2003. Elsevier Science.

- [162] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. Technical Report UIUCDCS-R-2003-2334, University of Illinois at Urbana Champaign, April 2003.
- [163] Koushik Sen, Grigore Roşu, and Gul Agha. Runtime Safety Analysis of Multithreaded Programs. In *Proc. of ESEC/FSE'03: European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, Helsinki, Finland, September 2003.
- [164] Koushik Sen and Grigore Rosu. Generating optimal monitors for extended regular expressions. In *Proceedings of 3rd International Workshop on Runtime Verification (RV'03)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*, pages 226–245. Elsevier, June 2003.
- [165] Koushik Sen, Grigore Roşu, and Gul Agha. Online Efficient Predictive Safety Analysis of Multithreaded Programs. In *Proceedings of TACAS'04*, volume 2988 of *LNCS*, pages 123–138. Springer, 2002.
- [166] Natarajan Shankar, Sam Owre, and John M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.
- [167] A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logics. *Journal of the ACM (JACM)*, 32(3):733–749, 1985.
- [168] A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logics. *Journal of the ACM*, 32(3):733–749, 1985.
- [169] O. Sokolsky and M. Viswanathan. *Workshop on Runtime Verification (RV'03)*, volume 89 of *ENTCS*. Elsevier, 2003.
- [170] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *STOC*, pages 1–9. ACM Press, 1973.
- [171] Larry Joseph Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic*. PhD thesis, Massachusetts Institute of Technology, 1974.
- [172] Scott D. Stoller. Model-Checking Multi-Threaded Distributed Java Programs. In *Proc. of SPIN'00: SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–244, Stanford, California, USA, 2000. Springer.
- [173] Robert E. Strom and Shaula Yemeni. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, January 1986.

- [174] K. Thompson. Regular expression search algorithm. *CACM*, 11(6):419–422, 1968.
- [175] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model Checking Programs. In *Proc. of ASE'00: International Conference on Automated Software Engineering*, Grenoble, France, September 2000. IEEE CS Press.
- [176] Pierre Wolper. Constructing Automata from Temporal Logic Formulas: a Tutorial. volume 2090 of *LNCS*, pages 261–277. Springer, 2002.
- [177] H. Yamamoto. An automata-based recognition algorithm for semi-extended regular expressions. In *Proc. of MFCS'00*, volume 1893 of *LNCS*, pages 699–708, 2000.
- [178] H. Yamamoto. A new recognition algorithm for extended regular expressions. In *Proceedings of ISAAC'01*, volume 2223 of *LNCS*, pages 257–267, 2001.
- [179] H. Yamamoto and T. Miyazaki. A fast bit-parallel algorithm for matching extended regular expressions. In *Proc. of COCOON'03*, volume 2697 of *LNCS*, pages 222–231, 2003.